NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ARTILLERY SURVIVABILITY MODEL

by

Yusuf Z. Temiz

June 2016

Thesis Advisor:	Christian Darken
Second Reader:	Michael Guerrero

This thesis was performed at the MOVES Institute

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704–0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington, DC 20503.			
1. AGENCY USE ONLY ***(Leave blank)***	**2. REPORT DATE** June 2016	**3. REPORT TYPE AND DATES COVERED** Master's thesis	
4. TITLE AND SUBTITLE ARTILLERY SURVIVABILITY MODEL			**5. FUNDING NUMBERS**
6. AUTHOR(S) Yusuf Z. Temiz			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) N/A			**10. SPONSORING / MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol number NPS.2016.0050-IR-EP6.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

This work investigates the use of modern simulation techniques for evaluating artillery movement doctrine. A simulation called the Artillery Survivability Model was created as a proof of principle. The simulation incorporates the most salient features relating to artillery survivability according to our small-scale survey of expert opinion on this subject. It consists of a 3D agent-based simulation that incorporates AI technology that is novel to this domain, including terrain analysis, advanced movement planning, and GPU-based particle filters to represent enemy anticipation of friendly artillery behavior.

The simulation has been created with the popular game engine Unity 3D, and has two different modes. The first is the experiment mode, which is executed from command line without rendering any image, and runs up to 50 times faster than the real-time simulation. Therefore, it is a suitable platform to perform multiple runs for experimenting. The experiment mode also enables users to set their own design of experiment by manipulating an editable CSV file. The second one is a real-time mode that renders a 3D virtual environment of a restricted battlefield where the survivability movements of an artillery company are visualized. This mode provides detailed visualization of the simulation and enables future experimental uses of the simulation as a training tool.

14. SUBJECT TERMS Survivability, Artillery, GPGPU, AI, Agent-based, Simulation, 3D, Unity, Experiment, Fast-forward			**15. NUMBER OF PAGES** 97
			16. PRICE CODE
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UU

THIS PAGE INTENTIONALLY LEFT BLANK

This thesis was performed at the MOVES Institute

Approved for public release; distribution is unlimited

ARTILLERY SURVIVABILITY MODEL

Yusuf Z. Temiz
First Lieutenant, Turkish Army
B.S., Turkish Military Academy, 2007

Submitted in partial fulfillment of the
requirements for the degree of

**MASTER OF SCIENCE IN MODELING,
VIRTUAL ENVIRONMENTS, AND SIMULATION**

from the

**NAVAL POSTGRADUATE SCHOOL
June 2016**

Approved by: Christian Darken
Thesis Advisor

Michael Guerrero
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This work investigates the use of modern simulation techniques for evaluating artillery movement doctrine. A simulation called the Artillery Survivability Model was created as a proof of principle. The simulation incorporates the most salient features relating to artillery survivability according to our small-scale survey of expert opinion on this subject. It consists of a 3D agent-based simulation that incorporates AI technology that is novel to this domain, including terrain analysis, advanced movement planning, and GPU-based particle filters to represent enemy anticipation of friendly artillery behavior.

The simulation has been created with the popular game engine Unity 3D, and has two different modes. The first is the experiment mode, which is executed from command line without rendering any image, and runs up to 50 times faster than the real-time simulation. Therefore, it is a suitable platform to perform multiple runs for experimenting. The experiment mode also enables users to set their own design of experiment by manipulating an editable CSV file. The second one is a real-time mode that renders a 3D virtual environment of a restricted battlefield where the survivability movements of an artillery company are visualized. This mode provides detailed visualization of the simulation and enables future experimental uses of the simulation as a training tool.

THIS PAGE INTENTIONALLY LEFT BLANK

DISCLAIMER

The reader is cautioned that the computer program developed in this thesis may not have been exercised for all cases of interest. While every effort has been made, within the time available, to ensure that the programs are free of computational and logical errors, they cannot be considered validated. Any application of these programs without additional verification is at the risk of the user.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF ACRONYMS AND ABBREVIATIONS

AI	Artificial Intelligence
ASM	Artillery Survivability Model
FFPAS	Firefinder Positioning Analysis System
GPGPU	General Purpose Graphics Processing Unit
IED	Improvised Explosive Device
IRB	Institutional Review Board
POC	Platoon Operations Center
SME	Subject Matter Expert

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank to my wife Burcu, who fed me and kept me healthy both physically and psychologically to achieve the formidable task of writing this thesis in such a busy and short schedule. And for sure, if it were not for my advisors, Chris Darken and Michael Guerrero, this thesis would not reach its final form. I also want to express my gratitude to them for their support.

Finally, being a loyal member of my Armed Forces I am thankful to the Turkish Nation who was my sponsor and gave me the chance to live in the beautiful city of Monterey and study in NPS for two years.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. PROBLEM STATEMENT

War is a complex phenomenon that happens unexpectedly and in a multidisciplinary environment. Since the ancient times, various plans and detailed preparations have been implemented by commanders to change the flow of the war and become the ultimate winner. In this fight, artillery has always been an irreplaceable element after its invention. Its importance dates back to 12th century China, where the first cannon was shot. Throughout history, several cannons and mortars have been designed to take advantage of the indirect fire.

The evolution of artillery triggered the development of position tracking systems, which are built to locate the precise indirect shooters. Today's equivalent of these devices, counter-battery radars are capable of detecting firing positions in minutes, which significantly decreases the survival time spent in a firing position. Therefore, artillery units have limited time to conduct their fire missions in the planned firing positions, because their locations are probably going to be detected in a short span of time. To be able to provide continuous fire support, which is one of the main purposes of the artillery, units should move to the next planned or available firing position for further shootings.

On today's fluid, crowded battlefield, the movement and positioning of field artillery units is a very complicated process that includes position selection, terrain management, movement planning and control, and the coordination of survey support for firing and target acquisition operations (Department of the Army, 2001). As the war goes on, the challenge of finding new positions will get even harder with time, since many firing spots would have already been occupied before, the attack of our own troops would not always advance, and artillery units could have been stuck on the same area. Detailed conduction of air, ground, and map reconnaissance may increase the number of planned movement and positioning maneuvers; but unexpected circumstances, and failed or delayed

offensive tasks would force the artillery units to occupy new, unchecked positions. This decision-making process under enemy fire would be stressful enough to lead to unsuccessful maneuvering and positioning.

In addition to survival, artillery units have to coordinate their movement for efficient fire support. Rather than changing whole battalion positions, it is wiser to have one or two batteries shooting while other units change positions. In this way, the continuous fire support need of maneuver units can be satisfied. Movement planning will become more complicated if the assigned physical area is smaller, or if the terrain restricts movement and canalizes troops to certain trails. Moreover, logistics considerations will affect the movement planning in order to have efficient resupply tasks.

There are numerous parameters and considerations to move and position artillery units properly. Even detailed and sensible mission planning before battle provides decision aid to a certain degree. During intense battles, the concentration on the firing missions, and communication and coordination with maneuver units are the real burdens that require a lot of effort. To enhance a mission's success, units should focus on positioning and movement coordination.

Ultimately, there is a need for a new algorithm, method, and tool, which can facilitate this decision-making process and help choose optimal destinations for future firing missions. Currently, there is no software tool in use that optimizes position selection and unit displacement for artillery. To fill this gap, the purpose of this thesis is to create a software tool that takes real terrain information, friendly artillery units capabilities, fire missions, enemy artillery shootings, enemy radar target acquisition capability, resupply tasks (partially), and other aspects into account. The following chapters will focus on the creation of this new software tool.

B. SCOPE OF THE THESIS AND RESEARCH QUESTIONS

This thesis focuses on creating a 3D agent-based simulation that computes the recommended survivability movements of an artillery unit to new firing positions in a given position area.

The thesis will focus on the survivability movements of self-propelled howitzer artillery unit that supports attacking maneuver units. The modern howitzers are able to compute their own technical firing data, such as T-155 FIRTINA (Turkish Army), and M109A6 Paladin (U.S. Military). The simulation will analyze the terrain for optimal firing positions, assign howitzer batteries, platoons, and sections to these positions, and will look for new proper positions when the previous ones are detected or are no longer available for accurate shooting.

Following are some assumptions:

- Forestlands and small areas with a high-level density of trees will be regarded as obstacles, and will provide concealment. Trees will be not included in the angle-of-site to crest calculations.
- A company-level artillery unit will be simulated with six self-howitzers in the 3D environment. It will be assumed that positioning battery support elements are properly done.
- Logistics with infinite supply will be assumed to be running perfectly. When the howitzer ammunition and fuel are depleted, one battalion rearm, refuel, resupply, and survey point (R3SP) will be simulated on a fixed position for howitzer platoons and sections to rearm. Damaged howitzers will fight until they are destroyed, rather than being sent to Field Trains or repaired.
- The batteries will consist of two platoons, formed by three self-propelled howitzers and one platoon operations center (POC).
- As it is assumed that enough survey control points have been installed and marked on the terrain, survey operations will not be represented in the simulation. The goal of this thesis is to determine the optimal movement pattern of an artillery unit in combat, by modeling limited battlefield environment. Although there are numerous threats for friendly artillery units on the battlefield, this thesis will focus on the enemy artillery fire as the main threat. The

simulation terrain will be imported from real terrain satellite maps, using a Bing (Microsoft) database.

- A Unity 3D Game Engine will be used to create the simulation.
- The Agent (howitzer) behavior will be modeled. Tactical path finding algorithms will be developed to specify the survivability movement.

Following are research questions:

1. Can an adequate algorithm for artillery be built for optimal survivability moves by modeling the terrain, weather, enemy howitzers and radars, and enemy target acquisition assets on a 3D environment?
2. How can an indirect fire threat map for artillery be built for further analysis and incorporation into the algorithm?
3. Can the simulation constructed to develop and validate the algorithm provide insight into the factors affecting the survival of the artillery?
4. By comparing their work and simulation results, can this simulation tool train artillery officers who would be asked to inspect the terrain, evaluate the tactical situation, and determine the positioning change path of artillery unit beforehand?

C. BACKGROUND

This thesis will cover movement considerations for survivability in conventional offensive missions. Defensive operations, urban warfare tactics, and other similar operations will be excluded from this study.

1. Main Considerations

One of the main positioning considerations is to avoid positioning artillery units close to major avenues of approach (Department of the Army, 2014). The continuous fire support should not be interrupted by any possible enemy breakthrough. According to the manual, commanders assign artillery units to a position area, where individual units can maneuver to survive. This area is not a specific location for the artillery to occupy. Instead, artillery commanders should choose appropriate firing positions in the position area coordinating with

maneuver headquarters, artillery headquarters, and fire cells. Prior to battle, picking up suitable positions, inspecting the whole position area, and putting them in a reasonable order for occupation in the following phases of war are crucial for proper reconnaissance.

In addition to checking suitability of the positions, possible routes among these positions have to be inspected for safety, cover protection, and ease of movement. Having an efficient plan depends upon a detailed planning process, which requires a dedicated amount of time. Although commanders usually have enough time to plan operations before any contact, time will become a more restricted resource in oncoming phases of war. Moreover, as the maneuver develops, the planning factors become more complicated, and a better coordination between the headquarters is required. To facilitate movement planning, aerial reconnaissance provided by helicopters or unmanned aircraft systems can be used.

During the positioning plan process, the range of friendly weapons must be taken into account, and the effective range should be maximized to provide more efficient fire support. Additionally, “positioning considerations include communications requirements, security risks, and logistical support” (Department of the Army, 2014, p. 1–52).

The vulnerability to detection due to firing signature can be weakened by being able to disperse, hide, fire, and then displace quickly. These actions will increase the chance of survivability and the survival time. With the advance of the technology of target acquisition assets, time that can be spent on a firing position is decreasing. This time duration must be defined elaborately by considering the trade-off between continuous fire support and survival. The trigger for displacement may depend upon the “number of rounds fired in current location, duration of firing, and time in position” (Department of the Army, 2014, p. 1–52). To summarize, some of the main things to consider during the coordination of the position area include:

- The capability to achieve essential fire support tasks (EFSTs) and essential field artillery tasks (EFATs) assigned to a battalion
- "Maximum range requirements and available ammunition to support the tasks" (Department of the Army, 2000, p. 3-2)
- Terrain suitability for offensive and defensive considerations
- Communications with higher- and lower-echelon units, as well as adjacent units
- Survivability
- Future operations (Department of the Army, 2000)

2. Tactics and Technical Details

Since this thesis will focus on the operation of modern and self-propelled howitzers, we will focus on procedures for the M109A6 Paladin howitzer, which makes the Department of the Army (2000) our main reference manual.

> A position area for artillery is an area assigned to an artillery unit where individual artillery systems can maneuver in order to increase their survivability. A position area for artillery is not an area of operations for the artillery unit occupying it. The maneuver commander assigns position areas for artillery as a terrain management technique. A position area for artillery potentially attracts enemy counter fire, so other units should stay away from that area to avoid enemy artillery attacks. The exact size of a position area for artillery depends on the mission variables of mission, enemy, terrain and weather, troops and support available, time available, civil considerations (METT-TC). (Department of the Army, 2014, p. 4–14)

"Under normal conditions, the smallest unit for tactical displacement is the platoon" (Department of the Army, 2000, p. 3-2), which is considered an individual march element that facilitates command, control, and logistical operations. "A Paladin platoon may require a position area on the order of 1,500 by 3,000 meters" (Department of the Army, 2000, p. 3-3). The higher the threat of counter fire, the more survivability moves are required by Paladin within a given position area. Controlled by one Platoon Operation Center; another option would

include a battery (six howitzers) operating on a single position area of 3,000 x 3,000 meters.

As an advantage over conventional howitzers, Paladins can occupy unsuitable firing positions that usually limit the movement and the fire capacity of older models. If there is enough space to establish an azimuth of fire, and the ground is firm enough to move around, the position can be considered as appropriate for Paladin operations.

> In a mid- to high-intensity threat environment, the Chief of Section must assume that the Target Acquisition assets of enemy are able to detect the first round fired from any position and that the enemy would respond in as little as 5 to 12 minutes. Paladin survives with the combination of movement and dispersion. A survivability move of 300 to 500 meters removes the howitzers from target footprint of most threat artillery systems. Managing survivability moves requires teamwork between the howitzers and the Platoon Operations Center. (Department of the Army, 2000, p. 3–11)

The tactical situation affects the employment method of howitzers. For example, if the enemy air threat is considered high, units should operate more decentralized, and howitzer units should operate as single howitzers in pairs, or at least as platoons. Using howitzers in little groups rather than operating as a whole battery (six howitzers), increases survivability, and enables the battery executing more than one mission type at the same time.

The battery Commander should also assess enemy counter-fire threat when deciding on the deployment method. In case of high enemy counter-fire threat and low ground attack threat, it may be wise to use howitzers in pairs. On the other hand, keeping howitzers together and letting them operate as platoon (three howitzers) or battery (six howitzers) will provide units with mutual air and ground defense. Additionally, as the level of the employment method decreases, the operations become harder to command and control, which requires a higher crew training level.

The platoon operations center usually does not change positions with howitzers on the position area. Instead, it relies upon cover and concealment to

survive and establish communication with upper units and howitzers. To avoid counter fire, it must stay outside the firing area. During occupation, POC follows the following procedures:

- Is there a suitable position? If yes,
- Does the position provide good communication?
- Does the position provide concealment?
- Are there any high-speed avenues of approach nearby?
- Does the position provide available escape routes (Department of the Army, 2000)?

> The ability to disperse, hide, fire, and then displace quickly helps negate the vulnerability of firing units to detection based on their firing signature. We should also keep in mind that this firing signature may also endanger any nearby units. The battery commander will issue movement criteria to the platoon leader for displacement and survivability moves. Some triggers for movement may include the number of rounds fired in current location, duration of firing, and time in position. (Department of the Army, 2014, p. 1–52)

Executing the fire plan during war and finding the next suitable firing position in the designated position area are complicated duties for artillery personnel, and often require additional considerations. As fighting intensifies, firing, moving, and occupying new positions are likely to get more intricate due to the lack of reasonable algorithms that provide adequate firing positions as an output. Making this decision without any computational aid would probably decrease the survivability rate and lead to unreasonable displacement movements.

To develop an efficient survivability movement algorithm that will increase the survivability time of friendly units, an unpredictable decision about the next firing position should be made after each firing. If our movement follows a predictable pattern, our units are more likely to be shot earlier, and have a shorter survivability time.

Creating an algorithm that chooses successive firing positions will provide our units with flexibility, agility, and time to respond to unexpected situations during battle. Almost every decision in war depends upon several factors such as weather, enemy, terrain, and duty. Comprehending these factors in a limited time, and reacting simultaneously to other problems, are heavy burdens for commanders. Using computational power will definitely improve reaction time to enemy fire and increase efficiency on the battlefield by choosing better firing positions, and by choosing them faster.

D. METHODOLOGY

Since the enemy will probably be able to predict our algorithm by using counter battery radars to analyze our position data, using an unpredictable pattern while moving between firing positions will require a more stochastic approach to define the algorithm. Since the human brain becomes more predictable with time, becoming unpredictable and making random choices is another reason to use a computational model.

The main purpose of this thesis is to create a simulation to overcome this decision dilemma by choosing the next firing position and generating a reasonable path to this position. Following are some specifications of the simulation:

- Real time
- 3D virtual environment
- Real terrain will be imported from satellite maps
- Agent based simulation

Depending upon numerous features that affect survivability time when our units decide to occupy this position, the simulation will generate suitable firing positions and score them. As mentioned previously, there are many considerations related to choosing a good firing position.

Rules and suggestions in manuals are often vague and do not follow math formulas. Moreover, they lack additional information that could easily be defined by an expert. To create an efficient position-choosing algorithm, interviews will be conducted with subject matter experts (SME). Thus, the lacking knowledge will be elicited from experts, and features of positions will be quantified and prioritized.

After interviewing SMEs, a cognitive task analysis will be made to design the conceptual model. According to the answers, factors that affect specific functions will be weighted and quantified. Factors considered as not so important will be excluded from the model. Interview results will aid in creating the conceptual model by benefiting from the ideas and the experience of experts. Another reason to consult experts is to make the simulation more objective, rather than promoting a biased simulation.

Because the simulation is agent-based, it will benefit from several Artificial Intelligence (AI) modeling techniques. Entities will have their Finite State Machines, will provide their own reasoning, and will act on their own. Random variables will be used on some parts of their decision-making algorithms to test different options, find relationships between these factors, and survivability time. The behavior of friendly units and enemy artillery will be modeled. To do this, several methods will be studied and examined to find the perfect match for the simulation's purposes.

After designing the simulation model, it will be coded in the Unity 3D game engine platform, which is a popular and free game engine coded in C# and TypeScript (a typed superset of JavaScript that compiles to a plain JavaScript language).

The Unity 3D community provides a lot of support for developers. There are a number of online forums where developers can find answers to their questions. It also has an online store where many 3D models and pre-written

scripts (i.e., “assets”) are sold or are free to users. Our simulation will benefit from these assets to save time and focus on the main problem.

At the end of the software building process, an experiment will be conducted to run this simulation multiple times. The experiment will be designed to determine a regression model about artillery survivability time. The factors that are assumed to affect artillery survivability time will be inspected by executing a two-level fractional factorial design experiment. To achieve this, a different approach will be used to make the simulation run faster, as multiple runs should be executed for each design point to get more stable results. Fast-forwarding the simulation will be problematic since the movement logic of agents will be waypoint-based. The larger the time step (delta time), the higher the danger will be to overshoot waypoints, which can cause oscillation around the waypoints. A detailed discussion about waypoints and the analysis tool JMP Pro 11 can be found in Section III.F.

E. LITERATURE REVIEW

There are several military simulations in use, both constructive such as Combat XXI and virtual such as VBS 3, which model environment, physics, fighting and firing (direct or indirect) etc. in different ways. These simulations make it possible for commanders to try new tactics without any physical loss or expenditure, and also for soldiers to train different aspects without extra effort. The focus of these simulations is on maneuver units and direct fire physics. Although the artillery is one of the major factors on the battlefield, either the specifications of contemporary computers do not allow simulations to contain elaborate AI for indirect fire units, or developers do not add details to the fire supporter’s model. There is no simulation that calculates the optimal paths to the next firing position. This study intends to fill this gap.

In this section, related researches will be reviewed. Because of the lack of studies similar to this, research that does not directly address the questions of this thesis will also be discussed. There is a lot of research directly related to this

thesis made by the Chinese Artillery Academy; however, because of their confidential level, it was not possible to access these from foreign servers.

The tool that is based on a very similar idea to the focus of this thesis, namely “finding the optimal positions,” is the Firefinder Positioning Analysis System (FFPAS), “a software tool that predicts the site-specific weapon location performance for Firefinder radars for a wide range of potential weapon placements and characteristics” (Fish & Murray, 2008, p. 30). FFPAS analyzes the terrain to find optimal positions for counter-battery radars that are used to inspect the trajectory of enemy artillery rounds to calculate the position of the enemy artillery. The simulation looks for suitable spots that could provide a wide range to detect enemy rounds by calculating Line of Sight (LOS). It is not a simulation based on artillery unit agents with AI.

“Optimization of a Marine Corps Artillery Battalion Supply Distribution Network” by Ryan R. Heisinger (2007) is about a model that simulates “a supply distribution network of roads between the battalion supply area, the firing batteries, and the headquarters battery” (Heisinger, 2007, p. 7). The model uses “Dijkstra’s algorithm to calculate the associated shortest travel distance between each pair of logistics nodes and then enumerate all possible tours through the logistics nodes” (Heisinger, 2007, p. 7), which does not seem to be efficient compared to an A* search. This thesis takes into account only the distances; its associated model finds the shortest path between nodes using Microsoft Excel interface without any 3D simulations.

Another thesis, “A Markov Model for Measuring Artillery Fire Support Effectiveness” by Dennis M. Guzik (1988), presents a Markov model measuring effectiveness of the indirect fire support weapon in providing fire support to a maneuver element. It also includes a basic movement decision strategy that is calculated with a constant probability of detection. Determining the time to change the recently occupied fire position is a part of this thesis. However, rather than setting detection probability to a constant, it would be more reasonable if it were set to a dynamic variable that is dependent on the number of the rounds

shot from that position, the time spent on the position, and the firing angle of the shot round.

Another NPS thesis, “A Command and Control Wargame to Train Officers in the Integration of Tactics and Logistics in a Field Artillery Battalion” by Michael W. Schneider and Anthony R. Ferrara (1989), presents a computer-assisted war game that provides “battalion staff officers some experience in dealing with shortcoming” (Schneider & Ferrara, 1989, p. 9). The simulation is created in Pascal programming language, and has a 2D interface. The main idea is to force the decision makers, namely the artillery officers, to consider numerous tactics and decide upon a series of command and control orders. In the end, the game provides feedback about the game process and performance evaluation. Since the software is older and lacks 3D virtual environment, this system no longer serves as an active simulation. Common features of this system included in this thesis are that they are both battalion level simulations, and both aim to train artillery officers. Nevertheless, the old project has no AI agents, and it functions only as long as the user provides input.

“A Cost and Operational Effectiveness Analysis for Future Artillery System in Korea,” by Chunsoo Kang, studies Cost and Operational Effectiveness Analysis procedures of the future artillery systems in South Korea. The study focuses on measuring “the operational effectiveness of the field artillery system by using a computer simulation” (Kang, 1995, p. 7). This simulation runs different scenarios, and by quantifying performance characteristics, it compares and evaluates each of the all scenarios’ outcomes. While analyzing the performance, this system takes into account the rate of fire and response time. Different rates of fire were applied on a dispersed and moving enemy, and the effect of 10 minutes of response time was inspected, which is the required time for an artillery unit to change its state from moving to shooting on a fire position. The author assumed that the enemy’s artillery has the same features as those of his country. This study examines the fire effectiveness of the artillery system of its time against North Korea using a simulation that is neither a training tool or a 3D

simulation. However, it functions as an optimization tool, which computes the fire effectiveness of possible guns of the future. Different than this project, this thesis seeks to optimize the survival time of the artillery units and to find optimal firing positions, rather than to study fire effects. Nevertheless, enemy artillery fire effectiveness would affect units of friendly tactics, movement, and survival time.

Another NPS thesis, “Development of an Artillery Accuracy Model” by Chee Meng Fann, “explains the methodologies that predict the trajectory and accuracy of unguided, indirect-fire launched projectile in predicted fire” (Fann, 2006, p. 7). In addition, it describes, “the methodology for including various factors such as drag and drift in the trajectory calculation” (Fann, 2006, p. 7). This study inspects the ballistics of howitzer shots and studies an accurate, indirect fire shot via a simulation. Again, this model is focusing on firing capabilities, rather than defensive factors.

Although there are many studies about artillery, none of them investigates survival and defensive factors by running a complete visual 3D simulation. The goals of this thesis are to implement an enhanced and altered A* algorithm to find optimal firing positions, and calculate efficient paths between these positions. Additionally, this software will serve as a training tool by letting the user estimate the suitable, correct firing positions and comparing the results with the results of the AI agent.

II. COGNITIVE TASK ANALYSIS

Modeling survivability movements of artillery units in a 3D environment is a challenging task. To create a reasonable and valid simulation, important considerations should be included in the modeling part. Since there are numerous factors and the computational power of recent computers is limited, not everything can be added to the simulation. Initially, complicated and bigger models may be considered as better, more accurate products; however, complexity will cause other problems during the following phases of production, which would be much harder to solve. Due to limited time and resources, this thesis will scope down factors and environment to an admissible level by conducting cognitive task analysis.

Cognitive task analysis is comprised of two main parts. First, interviewing SMEs to elicit knowledge from them via Critical Decision Method. Due to the complicated nature of the task, it has not been entirely documented in manuals. There are several rules dispersed in different field manuals of different command levels. The primary concepts were discussed previously in Section I.C.1; however, there is a lack of method to follow while making survival movements. By consulting SMEs and asking them case-based questions, we will focus on deducing hidden experience and knowledge from interview outputs.

Next, is the concept map of the task. Decomposing the main task into subtasks will facilitate analyzing this complicated mission. During the design phase of the simulation, this visual model will be an important reference. Some parts of the map will be broken down to be inspected in more detail.

A. CRITICAL DECISION METHODS THROUGH SME INTERVIEWS

Because of the artillary survivability movement process, a critical decision method will be used to elicit knowledge from SMEs via interviews. Difficulty in decomposing the survival task into physical and cognitive subtasks, and the lack of insufficient instructions on how to manuever and change position during battle,

are the main reasons for the need for the additional cognitive task analysis by consulting experts.

> Critical decision method is described for modeling tasks in naturalistic environments characterized by high time pressure, high information content, and changing conditions. The method is a variant of Flanagan's critical incident technique extended to include probes that elicit aspects of expertise such as the basis for making perceptual discriminations, conceptual discriminations, typicality judgments, and critical cues. The method has been used to elicit domain knowledge from experienced personnel such as urban and wildland fireground commanders, tank platoon leaders, structural engineers, design engineers, paramedics, and computer programmers. (Klein, Calderwood, & Macgregor, 1989, p. 462)

Conducting interviews with SMEs is considered a human subject experiment. The whole interview process has been reviewed by the Institutional Review Board (IRB).

1. Interview Questions

a. Descriptive Questions

- What are the main reasons that affect the survivability of an artillery unit on a firing position?
- What criteria should be considered while assigning a position area to an artillery platoon?
- How critical is the coordination between FSCOOR, S3, and the Battery Commander?
- How effective can the POC be in choosing the next firing position in the position area?
- What are the cues in the battlefield for your own artillery unit to be detected by the enemy?
- What are the best terrain characteristics for proper shooting and maneuver?
- What is the time limit for staying in the same firing position? What are the triggers for displacement?
- Should we avoid shooting with higher elevations since the time for the round to reach the target will increase?

- What should the features be for a proper firing position?
- In an intense battle, which type of command and control serves for a longer survival— Centralized or Decentralized?
- What are the main threats for artillery units?
- What are the defensive measures to reduce the probability of success of the enemy aircraft's target detection ability?
- What level of howitzer employing is more efficient to survive? (Platoon, Paired, Single)
- Should the whole battery change its position simultaneously? If so, under which conditions?
- If it was hit by enemy artillery, can we tag a firing position as detected? Is it wise to occupy this attacked position again? Why?
- While changing positions, which formations are suitable? (Column, Box, Line, Wedge)

b. Case-Based Questions

You are the platoon commander of three Paladin howitzers in an attack mission. Answer the following questions:

- You are conducting a fire mission. Three enemy rounds fall 100 m east of your platoon and your mission requires one more shot. What would you do?
- The attack of our maneuver units is not progressing. The positions that you occupied are mostly attacked by the enemy, and there are few planned positions left that are not occupied yet. What would your strategy be for the next displacements? Which considerations would affect your decision making?
- You are changing position. While you move towards your new position through an already attacked area, you receive an immediate fire mission. What would you do?
- While you are changing position, one of your howitzers breaks and loses the ability to move. What would you do?

2. Analysis of the Interviews

Three subject matter experts have been interviewed for the research. The questions about artillery survivability have been asked in one session. The sessions were recorded in audio files, which were then transcribed to be kept safe in a locked environment to protect records from any PII breach.

Rather than inspecting the whole interview sessions, we will focus on key parts of the conversations where SMEs stated their experience and pointed out remarkable points that are not well defined in the manuals.

After units entered the firing position, SMEs agreed that the time after units shot their first round is more important than the time spent on the position since the first entrance, because enemy counter-battery radar will locate the position units shot from. Still, we should consider that our units are being observed via different target acquisition assets all the time and they will be more vulnerable to artillery fire when they occupy a firing position and change their state from moving to stationary.

After the first round has been shot, it is only a matter of time expecting counter fire from the enemy depending upon their radar quality and preparation time to respond. Subsequent rounds would let friendly units spend more time on the firing position and staying at the same location will become more dangerous. One SME stated that the number of rounds shot from a position is highly dependent on the firing rate. They can shoot five rounds in five minutes or only one if they are really slow.

The training level of friendly units was defined as an important survivability factor by all SMEs. Trained crews required shorter preparation time to occupy a firing position, to fire from the firing position, and to leave the firing position. Additionally, one SME indicated that the trained crew would shoot with higher accuracy, causing the enemy to lose more efficiency and indirectly affect the survivability of friendly units. Additionally, trained crews can defend themselves better when they encounter ground attacks.

Whereas the impact radius of incoming rounds is defined as an important factor by two SMEs, one SME mentioned that the gun they were shot by provides more clues about subsequent shots. If we are taking mortar fire, falling rounds are supposed to be less accurate than artillery fire. Enemy artillery fire will be more effective on our units than mortar fire, which will affect our survivability rate.

Enemy radar quality is a huge factor on the survivability of units. SMEs expressed that depending upon the intelligence provided and the reaction time of the enemy shooting counter fire, we could adapt our total time spent on a firing position. For example, if the enemy reaction is poor and our maneuver units need consistent and immediate fire support, we can decide to remain on the firing position and shoot more rounds from there. And depending upon the quality level of the radar; detection accuracy, detection time, and covered angle fan of the radar, the enemy would shoot with higher accuracy that will definitely affect the survivability of our units.

In addition to the factors discussed, SMEs named survivability factors as dispersion width, armor quality of our howitzers, weather conditions, terrain conditions, the pattern they follow when changing position, and defensive measures taken on the firing position.

One common answer was that the position of the supply points would not make a difference for selecting the next firing position, since a Paladin has high maneuverability capabilities and reaching supply points within a small area is not difficult to achieve.

When units were not in need of immediate fire support, or there was no scheduled firing mission, SMEs preferred using a safe path to the next firing position. One SME mentioned that using a path that goes through an already fired region was unwise because it is easier for the enemy to shoot old targets in a shorter time since they already have the required calculations for that area. Another SME stated that it is more crucial to choose a random path to the next firing position because if the enemy realizes that our units are following a unique

pattern, by choosing paths, they can guess and fire at our path. On the other hand, another SME argued that using a covered path would decrease our chance to be detected by an enemy's air assets, especially as our units become more vulnerable to detection when moving and leaving traces on land.

All of the SMEs had the same idea about avoiding predictability on the battlefield. They suggested randomizing decisions when choosing the next firing position or the path to the next position. One SME mentioned that humans are actually not that good at making random decisions, and they tend to follow the same patterns that ultimately become easier for an enemy to solve. The SME provided the example of the enemy guessing the convoy movement of a friendly unit, and implanting IEDs on their path. Following the same and usual pattern, enables the enemy to guess possible locations that our units will have on their path.

The SMEs were not eager to visit the same firing position again, as they were aware that the enemy saves calculations, and often fires against previous targets. One SME argued that it is advantageous to shoot from a previous firing position. Other one expressed that if returned to, a previously shot area with craters and holes on terrain may affect soldiers' psychology bad and reduce their ability to fight ambitiously.

Even if they were moving through a previously attacked area, all SMEs chose to execute an immediate fire order by occupying the closest firing position once they receive the order. One SME stated that he would react and shoot immediately, unless he received intelligence that the area he was passing was too dangerous to stay and shoot from.

B. CONCEPT MAP

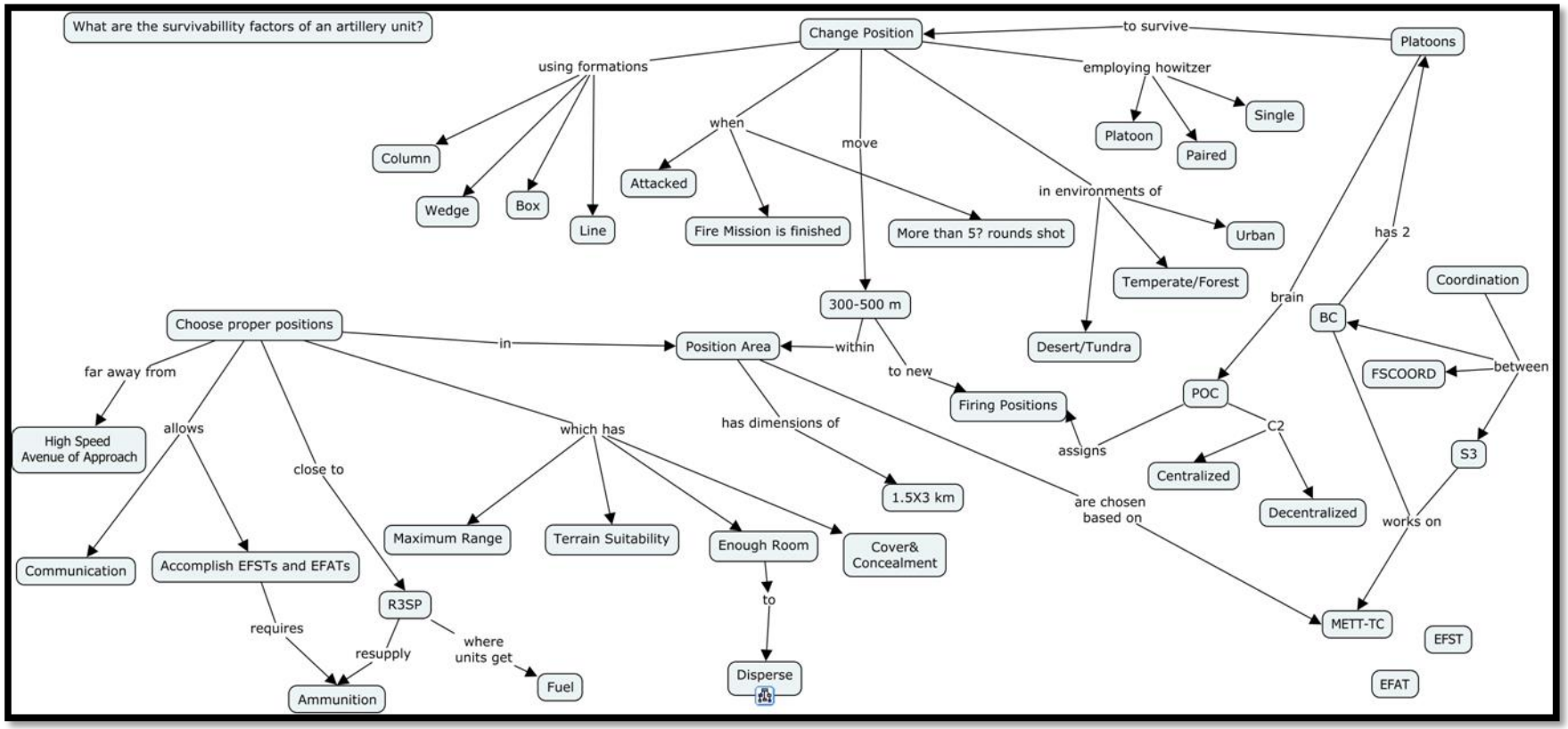
This study focuses on choosing proper firing positions and determining paths between these firing positions on a given position area. Factors that increase or decrease survival time will be studied. To serve to this purpose, after relationships between main concepts are determined, their subparts will be

analyzed and suitable factors will be modeled for the simulation. In addition, outputs of interviews with SMEs will contribute to this concept map by depicting sub-relationships of detailed concepts.

The concept map is a diagram that depicts the relationships between concepts. It is a tool that helps to understand the relationships between sub-concepts by visualizing them; hence, data and information visualization provides a more comprehensive and better understanding. Figure 1 provides a broad picture of the artillery survival system.

While working on the development of the simulation, the idea was to follow the map that defines the main concepts and sub-concepts. Using the information we received from the SMEs and having our concept map as a broad picture in front of us, we started to develop the software by creating small size applications that focused on different parts of the concept map. Additionally, the discussions we had with SMEs shaped the logic and the rules of the software. During the long hours spent on the development process of the simulation, our main concern was to find the answers to our research questions by modeling the right thing.

Figure 1. Concept Map



III. MODEL DESIGN

The interviews made with SMEs and the concept map substantially drew the borders of our model design. Having field manual instructions only would not suffice to solve our problems of finding a fair algorithm for survivability movements. Insights gained from the cognitive task analysis filled the gap of information needed on reactions against certain situations. Additionally, we wanted agents in the model to react rationally like an experienced artillery officer would do on the battlefield. Therefore, we referred to Chapter II while working on the design of the Artillery Survivability Model.

The Artillery Survivability Model (ASM) is an agent-based simulation created with the Unity game engine. Its main purpose is to visualize survivability movements of a platoon-level howitzer unit, and create useful decision-making algorithms for survivability movements to make them unpredictable and efficient. Terrain features, which are real terrain data imported from satellite maps, are processed to determine proper firing positions within the designated position area. Scores are then assigned to these firing positions depending upon several variables. According to these scores and the distance between them and the firing positions, agents pick up next firing positions from which to shoot.

Additionally, the ASM contains a position tracker algorithm that is particle filter-based and is executed on GPU. It is a similar version of the simulacra, which is explained in Darken's paper (Darken & Anderegg, 2008). This position tracker algorithm guesses possible locations of blue agents, when their position is revealed on enemy radar after they shoot.

For several reasons, the Unity game engine is preferred to create this ASM simulation. First, the author's familiarity with the software and C# programming language made it a strong candidate. Additionally, its popularity among game developers provides a vast public support for online forums and blogs that facilitate learning, adapting, and finding solutions. Unity also allows

building applications on almost every platform without needing additional code tuning. One of the other advantages is that Unity has an online asset store, where developers can sell or publish their 3D models, scripts, and solutions. Many of them are easily integrated and some provide fast solutions for complex problems. We used both free and purchased assets from the online store for terrain, pathfinding, and explosion effects.

The 3D howitzer models are from the Delta 3D open-source game engine library, which can be found at SourceForge (a web-based service that offers a source code repository).

In this chapter, the main components of the simulation and the structure of the model design are described. Information about how they are created and which logic steps have been used in their development are explained with minimal detail. Additional information about some crucial points and algorithms can be found in Chapter IV.

A. BASICS OF THE TERRAIN

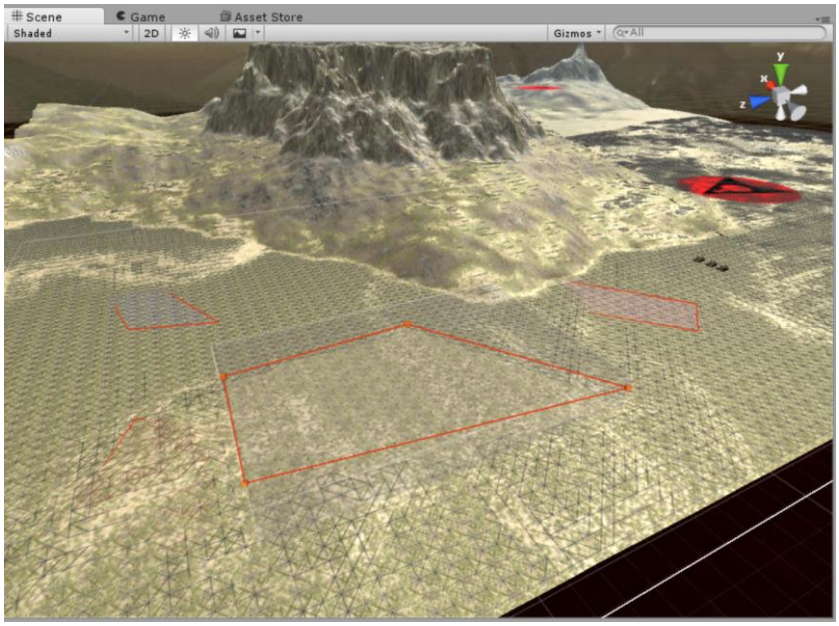
ASM uses the popular unity assets “Terrain Composer” and “World Composer” written by Nathaniel Doldersum. The Terrain Composer is a plugin to generate terrain with various heightmaps, splatmaps, trees, grass, and additional objects such as rocks, which can be considered as obstacles. The World Composer is a great tool to import real-world heightmap data from the “Bing” Microsoft database, and has additional features such as determining tree positions by processing satellite pictures.

Our model uses real-world terrain information by importing the usual training field of artillery units to the 3D environment using the “Terrain Composer.” The positions of the trees will be defined by processing satellite image data. A detailed explanation of the Terrain Composer can be found in the Chapter IV. Other features of land will be edited manually via the Unity 3D editor by assigning areas for different terrain types. Mainly, there are six types of terrain

defined: solid ground, rocky ground, sand, mud, soft ground, and non-walkable terrain.

Areas with distinct features are defined in the Unity 3D Editor by adding a “Graph Update Scene” object to the scene (see Figure 2), which is a part of the library “A* Pathfinding Project Pro.” Borders are assigned by selecting Shift-Click on the terrain. Nodes that fall within this designated area are assigned tags with many defined terrain types. Depending upon these manually assigned areas, specific splatmaps will be applied as texture, and nodes within these areas will be given corresponding penalties. Therefore, agents will prefer to move on appropriate terrain when their paths are calculated by the A* algorithm.

Figure 2. Graph Update Scene Object



Locations of artificial obstacles such as buildings are marked as non-walkable terrain, as well as dense trees that do not allow howitzer movement.

Trees play important roles in the simulation. After their positions are determined and forest areas are excluded as non-walkable areas, trees are regarded as one of the main survival elements of artillery because they cover

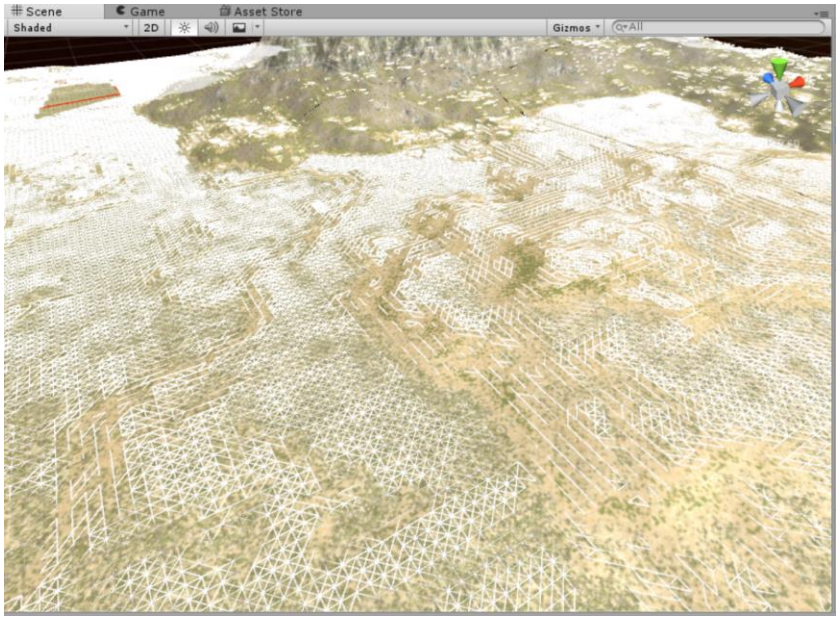
ground troops from the enemy's aerial surveillance assets. Every node has an attribute of "number of close enough trees," which is calculated by counting the trees in a specified radius from the node's location. This value is the main factor that determines the path of agents, because they are searching for a covered and fast path to the target. Details of A* pathfinding are described in the following sections.

B. GRID NODES

ASM uses a popular Unity asset for pathfinding, named "A* Pathfinding Project Pro," written by Aron Granberg. This asset serves as an additional plugin and provides fast and good structured A* pathfinding solutions. It has the ability to create different kind of graphs such as grid, navmesh, point, and recast graphs. These are the main data ground for agents to search paths to targets. We used grid nodes, since it is easier to store required data on each node, to process and manipulate data. The nodes are spread uniformly on the terrain. Another advantage of the grid graph is that runtime changing of graph needs lower computational power than other graphs.

Grid graphs (see Figure 3) are built by simply ray tracing from some distance above the terrain at pre-determined intervals. Our grid nodes have eight neighbors, to the cardinal and inter-cardinal directions. Walkable directions are visualized by straight lines between nodes. Additionally, nodes that are on a spot with a slope higher than a limit are tagged as non-walkable.

Figure 3. Grid Graph



A different grid graph is assigned to each agent group, namely a platoon-level artillery unit consisting of three howitzers. The leader agent searches for reasonable paths only on its assigned grid graph, which can be considered as the position area of the platoons (usually 1.5X3 km for a Paladin platoon). While the platoon leader is searching for a path to the assigned target, flank members of the platoon are searching paths to the offset locations of their leaders depending upon their type.

ASM has four grid graphs. Two of them are the position areas for each platoon, while the other two serve for calculating and caching paths between firing positions to be used on GPU side calculations. The first two main grid graphs are more detailed and store additional data on each note, while the final two have lesser resolution.

A* Pathfinding Project Pro assigns unique integers to each node that is connected and described as an “area.” Within a grid graph, it is possible to have different areas that are not connected via walkable nodes. We determine the area with the maximum number of nodes as a possible location for firing

positions and maneuver area for howitzers. Other areas will be not inspected since agents are not able to move and maneuver on them.

Calculations about the grid graphs are made offline, because they require a significant time. This data will then be loaded at the beginning of the simulation from stored files.

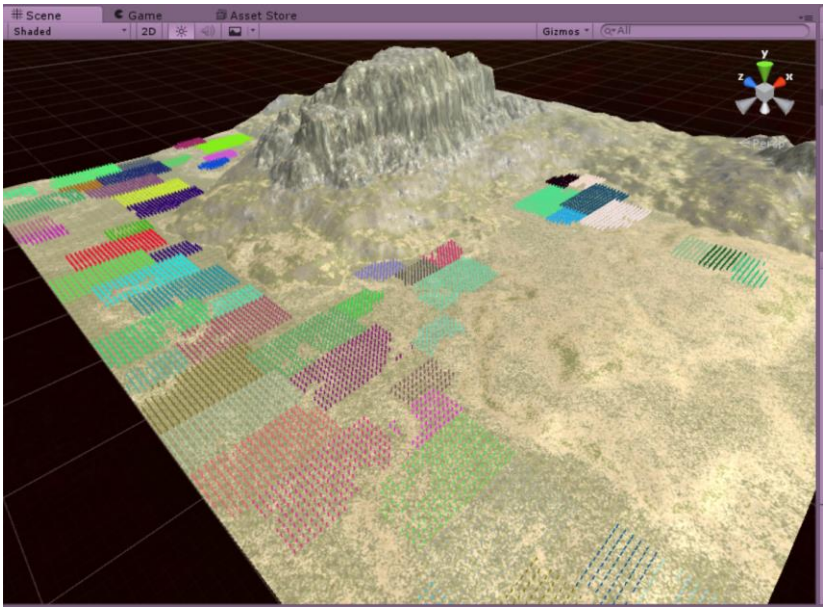
C. FIRING POSITIONS

Processing terrain data and determining firing position after the grid graph is created and nodes are tagged, their corresponding terrain type, and groups of connected nodes (that are able to provide enough space for howitzer platoons), will specify firing positions with different sizes.

Two different sizes are specified: “big” and “small” firing positions. Big firing position areas allow units more space to disperse and increase their survivability. The maximum area size on the grid graph is then filled with these two sizes of firing positions with an algorithm like flood-fill.

The main maneuver area is the maximum area size that has the most connected nodes, and is marked with the firing positions. To fill the area with rectangle firing positions efficiently, we searched through efficient rectangle filling algorithms. However, none of them did work perfectly, because our position area was not a rectangle. So we determined two 3D rectangular prisms with two different scales of width and height. These game objects were transformed to every walkable node that was not already tagged as a member node of a firing position and had higher walkable neighbors than four. Therefore, the center of the 3D object stands exactly on the position of the grid node, and the nodes whose positions were within the borders of the 3D objects were counted. If the number was greater than a threshold, the center node was checked as the firing position, and the other included nodes were marked as members of the firing position. In Figure 4, every firing position is marked with a different color.

Figure 4. Colored Firing Positions



The main criteria of choosing the next firing position in the simulation depends upon the quality of the position and the distance between the current position of the agent and the firing position. The score (see Figure 5) of the firing position is defined via a linear equation whose factors are:

- Number of trees on that firing position
- Mean of the slope
- Minimum quadrant elevation from the center node
- Number of visits
- Mean of the terrain type

The number of trees increases the total score linearly to a limit because they are regarded as main helpers to disguise friendly units from the enemy's eyes. If this number passes a threshold, it is set to zero having no positive or negative affect on the equation. The factor will be zeroed after a point when the number of trees no longer helps; on the contrary, they will slow down the movement speed and limit firing abilities of howitzers. Areas with a high number

of trees (i.e., forests), are already marked as obstacles and non-walkable regions.

The mean of the slope of the firing position's ground is another multiplier of our score equation. The mean of the slope of every firing position is calculated by checking y-values of the normals on the node positions for each member, and then taking their mean. They are scaled between 0 and 1, mainly because normals are normalized vectors. If the y-value is 1, the spot the surface is flat, which is better and more consistent for the artillery to shoot, and decreases the preparation time for shooting. Therefore, the optimal mean of the slope should be close to 1.

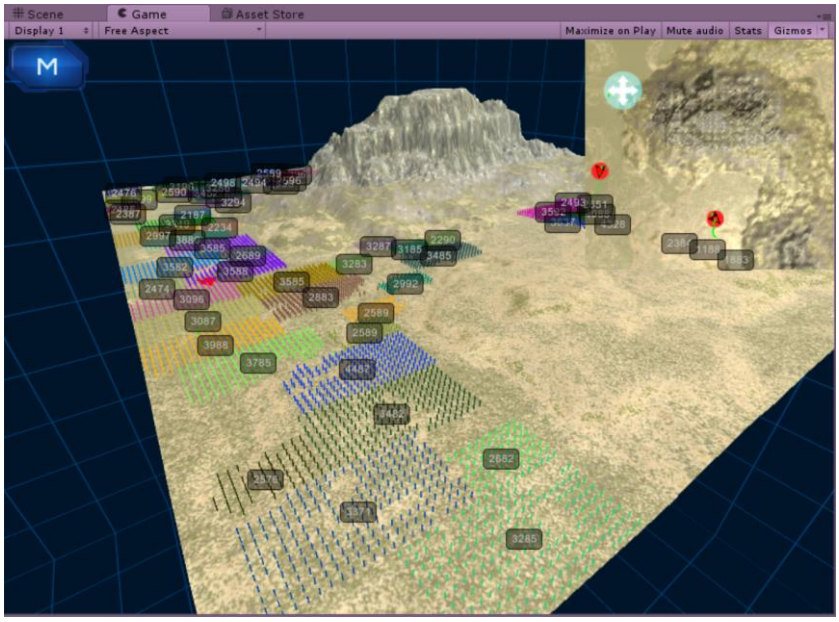
The minimum quadrant elevation affects shooting abilities of artillery. Often obstacles like hills between the howitzer and the enemy prevent them from being seen. However, if the obstacle is close and too high, the howitzers will need to elevate their barrels to higher levels to shoot. This will cause the shot round to follow a longer route to meet the target with a lower speed, thus making it easier for the shot round to be detected by enemy radar, as it remains longer in the air. This is an unwanted situation for artillery units, because they try to lower the chances of being detected at every opportunity. Therefore, an intermediate value is preferred, which is not too small and higher than a limit.

The number of visits to firing positions has a negative effect on the total score. The score was not zeroed when the position was occupied, because if the position was in a location with a high score for shooting units, we might want to save it for later. If each position has been visited at least once, units would need to compare firing positions by maintaining available scores.

The mean of the terrain type on the firing position is determined after checking the terrain type tag for each node and taking the mean, to verify solid ground for consistent shooting. The terrain type order of preference (from best to worst) would be: solid ground, rocky ground, sand, muddy, and soft soil with correspondence scores of 5, 4, 3, 2, and 1. The score of a firing position would

be higher if the firing position consisted of more nodes tagged with preferred terrain types. The mean score is then multiplied by 200 and added to the total score of the firing position. The terrain types also have an effect on the A* penalty calculations, which will be discussed in the following chapters.

Figure 5. Scores for Firing Positions



Apart from the firing positions of friendly units, the enemy evaluates the terrain and determines possible firing positions. However, different from the methods of friendly units, the enemy does not take terrain types into account, because it has limited access to the terrain on which friendly units are maneuvering. The enemy has recon assets such as satellite maps, aerial photos, and 2D military maps. However, without touching the real terrain, it is hard to determine the suitability of the ground quality for artillery fire, and to detect non-walkable terrain or obstacles that do not appear on maps or via air recon. As a result, their firing positions will not match with the firing positions of the blue forces, which cause errors in guessing the next possible firing position of friendly units, (an acceptable situation in real conditions).

For every grid graph, the firing positions are calculated and saved to a file in binary format to be loaded when the simulation is executed.

D. FINITE STATE MACHINE

The main AI of the agents is written in Finite State Machine structure. A popular asset, “Playmaker” by Hutong Games LLC, is used. This asset provides an easy editor interface, where the user can click, drag, and type the states to define the relationships, actions, and transitions between states. Additional complex actions are written in C# scripts.

Agent actions are driven by fire orders. A list of fire orders in CSV file format is provided to the application beforehand. It consists of numerous orders that musts be fulfilled at specific times. A typical fire order in the simulation has five variables: company name, distance to target, ammunition type, shot number by round, and scheduled time. If the fire order is not scheduled, its scheduled time is set to “-1.” These fire orders are processed at the company level, and assigned to the platoons to be executed at the specified time.

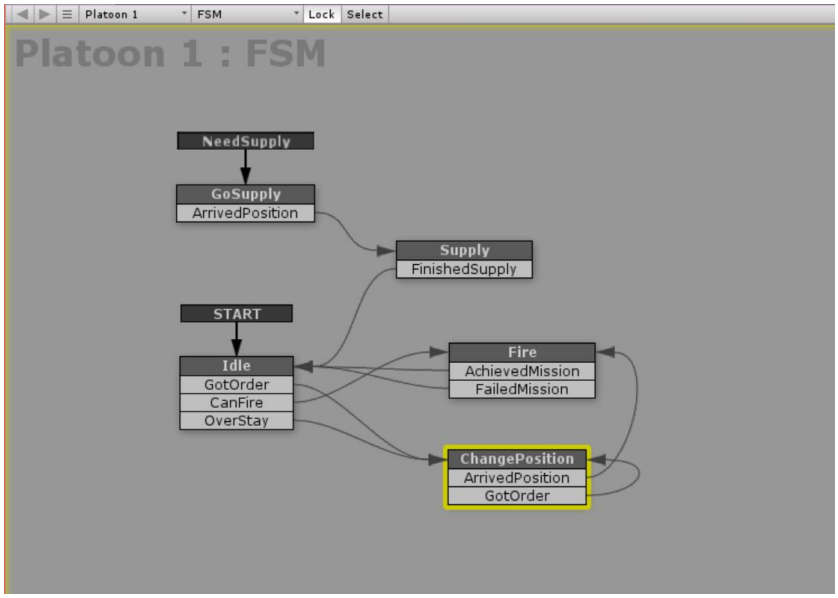
The company-level agent evaluates fire orders, puts them in priority, and assigns the most urgent fire order to an available platoon. It checks scheduled times and assigns an order if the platoon is in “idle” state. If both platoons are in a state other than “idle,” it looks to the total number of already assigned orders and assigns the order to the platoon with the least orders. If the scheduled time for an order is already passed, it removes this order from the order list and marks the order as “failed.”

The core AI of the simulation is the platoon-level Finite State Machine (FSM), where decisions are typically made. See the FSM structure in Figure 6. At the beginning of the simulation, the platoon reacts as soon as it receives fire orders from the company leader. It looks for a firing position to execute the most urgent fire order by sorting them in decreasing priority order. If there is a scheduled fire order, which has to be executed after a short time and is not at the top of the list, it is evaluated and scored, then placed in high priority to be

executed immediately. The top order in the fire order list becomes the goal order, and a suitable firing position is selected to carry out the mission. Every firing position's score is calculated again, and the list of firing positions is sorted in decreasing score order. The firing position with the highest score—located further than a lower-bound distance and closer than an upper-bound distance—is assigned as the goal for the platoon leader agent. The leader agent starts searching for A* paths to the target at each specified second interval, while the platoon's state changes to "ChangePosition."

The platoon moves to the assigned firing position until the leader agent moves closer to the center of the firing position. This event triggers the state changing to "Fire" state. First, flank members get to their positions by dispersing on the firing position and rotating the howitzers towards the firing direction. When they are ready to fire, (determined by preparation time of the unit and their training level), the desired type of rounds are shot from the designated number of howitzers. The time to prepare for shooting the second round will be shorter, since the units will just be reloading their guns. If two rounds are necessary, two howitzers with a higher amount of the specified rounds are fired. After rounds have been shot, the total number of their type stored in the howitzers is decreased by one. If the total number of round types falls below a number, a global event "NeedSupply" is called. Regardless of the platoon's state, it is automatically updated as "GoSupply."

Figure 6. Finite State Machine of One Platoon



The location of the supply point is determined when the environment is edited in Unity Editor. It is located on the far side of the terrain from the enemy. In "GoSupply" state, the leader travels to the supply point, and when it reaches the goal, the platoon's state changes to "Supply," during which new rounds are loaded into the howitzers. One of our assumptions is that the howitzers do not deplete their fuel before their ammunition. Therefore, fuel is not modeled in the simulation, and we assume that the howitzers resupply their fuel while their ammunition is being loaded. At some point, the supply process ends and the state again changes to "Idle."

If the platoon is not already on a firing position and has fire orders to execute, it looks for a nearby suitable firing position and moves to the location to perform the top fire order. After firing the first rounds from the currently occupied firing position, the platoon leader evaluates some factors to decide on the next step. The more frequently the platoon changes firing positions and fires, the higher are its survivability chances because being mobile makes it harder to be shot by the enemy. At the same time, the platoon leader considers executing fire

orders at their scheduled time. After the rounds are shot, the state changes to “Idle.”

Triggers to change position are the number of rounds shot from the firing position includes time spent on the same firing position, and the danger level of enemy fires. If the maximum number of rounds that can be shot from the same firing position has not been met, and the time spent on the firing position has not passed the maximum limit, one more group of rounds are shot from the same firing position, changing the state to “Fire.”

After shooting another group of rounds, FSM will fall into “Idle” state again. Checking the variables, the platoon leader ends up choosing either to fire another group or changing its position. The howitzer platoon carries out the same logic pattern until all of its members get shot and lose their ability to move, or are completely destroyed. When the efficiency of a howitzer unit falls below a value, it loses moving ability and starts to emit white smoke to symbolize that it has lost the ability to move. In this state, the howitzer cannot move, but can continue to fire. If the howitzer is the leader agent, another alive and mobile member takes over the duty of being the leader agent. The non-moving agent rotates the howitzer to the firing direction and follows fire orders by shooting from its position until it is completely destroyed, which is represented with red smoke in the simulation. When all three members of the platoon lose moving ability or are destroyed, the application records the time and restarts.

E. ENEMY AI

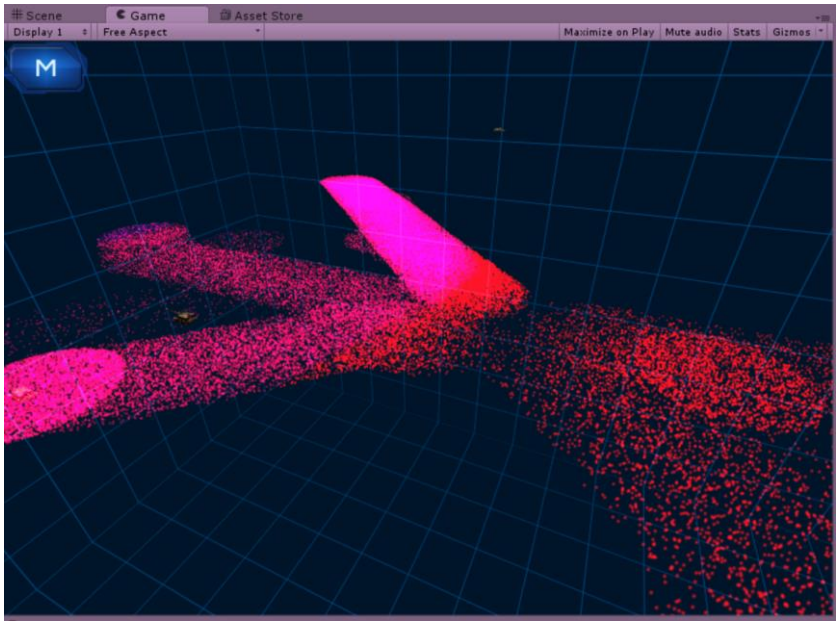
The AI of the enemy plays an important role in the simulation. Its main components are guessing the friendly unit’s position and shooting counter fires to these positions. After the friendly units shoot, their position is revealed on the enemy radar at a random time determined by the quality of the radar.

In the simulation, particles are used to model the movement of the blue forces. Since the required quantity of particles is high and the CPU is already busy with other calculations, we preferred to use a GPGPU (General Purpose

Graphics Processing Unit) programming to ease the burden of the CPU. Detailed information about GPGPU algorithms is provided in Chapter IV.

Guessing possible locations of the agents after their positions have been revealed on radar is achieved using particle filters. At the start of the simulation, 100,000 particles are created. Every particle has a variable of a float value that corresponds to the probability of friendly agents being at that particle's location. Additionally, they have attributes such as velocity, speed, color, offset, path number, and path number counter (See Figure 7).

Figure 7. Moving Particles



When particles are created for the first time, they are assigned random values of speed between minimum and maximum speed values of a howitzer, to represent every possible speed on the terrain. They are also assigned an offset value that is the distance between the center point and a random point in a unit sphere. This offset will always be used through all the movement calculations to visualize an area. The default probability is 5 per each particle, which totals a 500,000.

We first tried to assign a normalized mean value to the particles such 1/100.000; however, the constraint of writing to the shared memory at the same time, while iterating through all particles in parallel processing, pushed us to use integer values rather than using normalized float values for the probabilities of the particles. The built-in function we used to synchronize parallel treads, and which allows writing on the shared memory, “void InterlockedAdd (in R dest, in T value, out T original_value)” (Microsoft Shader Model 5), is able to write only to integers and unsigned integers. Therefore, the total probability of particles is 500,000 rather than 1.

To move particles on the GPU side without causing any collision, we cached every possible path in three-dimensional vector arrays: Assume that there are 50 firing positions and that for every firing position, 49 A* paths to other firing positions are calculated and stored offline. These A* paths are calculated on lower resolution grid graph (grid size: 20), not on the main high resolution grid graph (grid size: 5). Again for every firing position, one long vector array is created by adding paths to other positions together in a specified order. Therefore, on the GPU side, they can be read easily from this long array by using indexing technique.

Particles are visualized via post processing and are drawn after the entire scene is drawn. They first appear in simulation when the enemy detects the position of the blue forces for the first time. Then at every frame, particles are updated on the GPU side.

Particles are reset every time the position of the blue force is revealed on the enemy radar. This reset function is running on GPU as well. We first tried to reset particles on CPU side; however, manipulating all the data (100,000 particles) on the CPU and then trying to load them to the GPU caused a one to two second delay, thus slowing down the simulation. The main struggle was to create the logic of choosing the next reasonable firing positions for the blue force. So the enemy AI calculates the next positions by assigning weighted probabilities to the nearby positions on the CPU side, and by creating an array of the length

100, which consists of possible next position IDs that are repeated depending upon their possibility rate. If the position number 15 is likely to be visited 50% by the blue force, number 15 appears 50 times in that array. On the GPU side, a basic uniform random picker algorithm chooses from among the members of this array. Assuming that one three-dimensional vector holds three floating numbers that are each 4 bytes, 1200 bytes of data will be handed to the GPU rather than sending an entire particle dataset that is almost 6 megabytes (each particle possesses data of 60 bytes).

After particles are assigned path numbers, their next waypoint is determined by checking cached path array by index number, and when they are close enough to their target, their path number counter is incremented by one. Next, they choose the vector from cached path array again until they reach the end of their path.

When the enemy is ready to shoot, it looks for the highest probability holder within the “checkpoints,” which are positions stored in arrays. They consist of every firing position and every waypoint on the cached path array, which stores every possible path combination between firing positions. They are checked by summing up the probabilities of the particles within a specific radius. The highest total probability holder checkpoint and other highest two checkpoints—father than a determined distance—are chosen as three possible places on which the blue forces are likely to be present. Then, from these three places, one position is picked as the random firing position. While parallel looping through 100,000 particles on the GPU, particles who determine themselves to be close enough to a checkpoint attempt to increment the total value of the checkpoint on the shared memory by its probability value. However, because this process is occurring simultaneously, some particles would attempt to write on the same shared memory at the same time. To fix this problem, a built-in function of the Microsoft Shader library is used to coordinate and synchronize threads without causing any important delays. One limitation of this function is that it allows adding values only to integers or unsigned integers.

Once the enemy decides on the position and shoots toward that location, the blue forces in the effective range of the rounds are damaged depending upon the distance between them and the center point of the fallen rounds. The closer they are to the impact point, the more they are damaged.

Another process running on the GPU side is the culling function. The enemy has the ability to recon the terrain occupied by friendly forces via satellite maps and aerial photos. Whenever the enemy manages to take a photo of an area, if no blue force can be detected there, the probability of the particles wandering on that area is decreased. Determining that there is no counter-force on the scanned portion increases the probability of the presence of the blue forces in other regions. Therefore, the probability values of excluded particles are increased by the same amount, so that the total probability stays the same. This ability to scan the terrain of the blue forces and acquire intelligence is an indicator of the quality of enemy's target acquisition assets. During the experiment, this variable will differ by the quantity of reconnaissance attempts depending upon recon assets quality.

F. FAST FORWARDING FOR THE EXPERIMENT

ASM is an agent-based real time simulation. Movement of objects heavily depends on the time passed between two completely rendered frames, which is called delta time. Because the delta time is a multiplier of our movement distance logic, no matter what the frame rate is, two different simulations running on different hardware will have agents moved in the same distance in one second. A computer with a low-frame rate will have bigger delta time values and end up having the same values as a fast processor that has a higher frame rate. Using delta time ensures consistent visual outputs on different frame rates.

An agent's movement is waypoint dependent. After their A* path to the goal is calculated, they are assigned a list of waypoints to follow. They start moving by setting the first waypoint as the target. Basically, when they get close enough to that waypoint, they change their target by choosing the next waypoint

in the list. If the frame rate is too low, it will cause a higher delta time value, which will lead to big jumps on the movement at every frame. Having big distance jumps is dangerous for several reasons. First, the agent can move through a wall or obstacle without colliding. That visual artifact will appear as a teleportation effect. Additionally, the agent will oscillate around a waypoint, while trying to come closer than a specific distance to assign the next waypoint as the target. Having low frame rates and high delta times, will contribute to inconsistent movements and unwanted outputs.

To design an experiment and analyze variables that are affecting the survival time of the artillery, the ASM should be run several times. More runs will provide consistent means and lowering errors. Assuming that we run ASM in real-time mode and every run takes above 10 minutes, we would need weeks to finish every multiple simulation run on every design point. Therefore, we decided to fast forward simulation to be able to speed up the experiment process. Unity 3D has a built-in Time class, which has an attribute named “timeScale” that allows the user to speed up their applications by multiplying the “Time.deltaTime” value with that scale. This option is limited by a scale of 100; however, in our simulation, after incrementing the time scale above a critical value, agents start behaving inconsistently for the reasons described in the previous paragraph.

“V-sync is short for vertical synchronization. The computer screen refreshes a certain number of times a second. V-sync is an option that ensures that the frame buffer is filled as only as fast as the screen can read it. This prevents artifacts like tearing, where the frame buffer changes as the data being written to the screen causing a visual tearing effect” (Schuller, 2011, p. 158). By default, V-sync option is enabled in Unity 3D. Closing V-sync option boosts up frame rates from 60-80s to 140-200s when rendered in decent resolution with high quality textures. Again, having a higher frame rate does not speed up the simulation but provides a smooth video quality and animation flow.

Standalone Windows, Linux, or Mac OS applications built with Unity 3D can be called in batch mode via the command line. Running the application with

“-batchmode” tag, lets it run in the background without rendering any 3D image. In other words, the simulation actually runs, but the graphic card does not render. Another feature of Unity is that the user can see profile data in Unity Editor while the application runs in standalone player, via using ports sending messages between applications. Using this feature, we measured that our simulation reaches up to 10,000 frame rates in batch mode, meaning that every second the application manages to compute 10000 update cycles. To benefit from this frame rate boost, we developed a new mode for the simulation named “experiment mode.” In this mode, we fixed the delta time value at each update, rather than using the time passed until the last frame has been drawn. Using a secure fixed time value, which doesn’t cause any overshooting of waypoints or passing through solid objects, allows us to speed up the simulation up to 50 times in batch mode.

In the simulation, several functions are called after some time delay in seconds. In the experiment mode, we created additional time logic to deal with the waiting process. (Unity 3D uses coroutines principle to delay function calls.) The detailed scripts can be found in Section IV.C.

THIS PAGE INTENTIONALLY LEFT BLANK

IV. TECHNICAL DETAILS

While describing the main parts and modules of the simulation in Chapter III, we kept it simple in terms of providing a more understandable description of the model. The objective of creating a reasonable survival movement algorithm has been achieved, and its logic has been well described. In this chapter, we will provide detailed information about the methods and the compiled code. Mainly, three important components of the simulation will be examined here: A* path-finding calculations, parallel processing on the GPU, and speeding up the process of the simulation.

A. A*

A* is an efficient, widely-known and fast path-finding algorithm. Instead of implementing our own methods, we used an A* library, which can be purchased from the Unity Asset Store. Its name is “A* Pathfinding Project Pro,” and is written by Aron Granberg. It also has a free version that comes with limited features. At this point, we will not describe this feature in detail, but instead, we will inspect how we used this library to serve our purposes and where we used the A* methods.

Throughout the simulation, A* calculations are performed for two purposes. First, the movement of the agents and howitzers are bound to A* paths. They are being frequently calculated starting from the beginning of the simulation. Every second, two A* star paths are calculated per each agent. Moving in formation guided positions, as the leader agent computes A* paths to the target location, other members of the platoon that take position on the flanks compute paths to the offset positions.

The A* project uses Euclidean distances for heuristic cost. For a typical heuristic cost, it takes distances between grid nodes into consideration, if no additional costs have been assigned to nodes. In the simulation, we assign different weighted penalties dependent upon the terrain type and the number of

trees around that node. To make locations appealing and least cost driven, every node has to be assigned a default penalty score and those nodes in the desired area should have assigned lesser scores or “0” score.

We defined six different main terrain types for the simulation: solid ground, rocky ground, sand, muddy ground, soft ground, and non-walkable terrain. Assuming that trees cannot grow on sand and rocky ground, we specified three categories to define tree density on a location: low, medium, and high. Combining features of terrain type and density of trees on the location, we defined 15 different types (see Table 1).

Table 1. Terrain Types

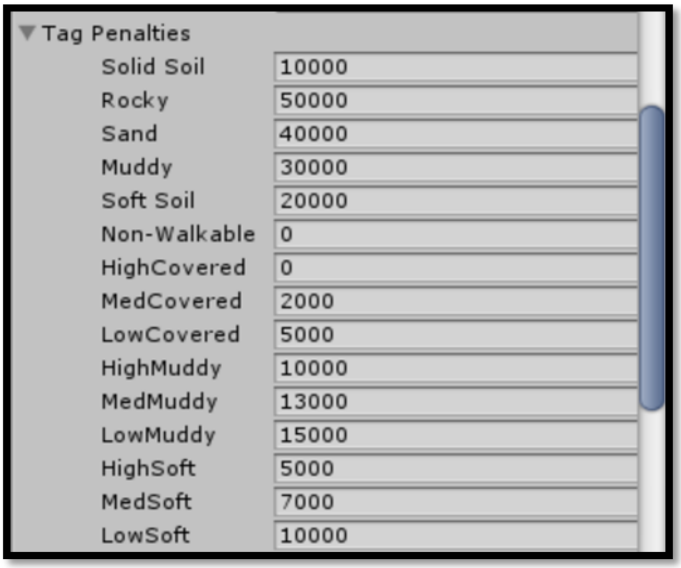
Nr.	Type	Number of Trees
1	Solid Soil	<3
2	Rocky Ground	-
3	Sand	-
4	Muddy Ground	<3
5	Soft Soil	<3
6	Non-Walkable	<3
7	High Covered Solid Ground	>15
8	Med Covered Solid Ground	>7
9	Low Covered Solid Ground	>2
10	High Covered Muddy Ground	>15
11	Med Covered Muddy Ground	>7
12	Low Covered Muddy Ground	>2
13	High Covered Soft Ground	>15
14	Med Covered Soft Ground	>7
15	Low Covered Soft Ground	>2

After specific locations are tagged with terrain type tags, offline calculations are made to determine how many trees are closer than a specific distance to the each node. Tags of the nodes that are already tagged as solid, muddy, or soft ground, are changed by adding “highly covered,” “med covered,”

or “low covered” words in front of their tag name, if they meet the requirement for the minimum number of trees around (see Table 1).

The main reason for tagging the nodes by offline computation is to assign penalties to each of them depending upon their tag name. Each agent in the simulation is attached with the C# script “Seeker,” which is a part of A* Pathfinding Project that calls A* path calculations. We assigned the same set of penalty scores to each agent (as seen in Figure 8), which are actually experimental numbers. We simply put terrain types in order of decreasing desirability and start to assign penalties as multiples of thousands in increasing order. The default cost of one unit node distance is 1,000 in the A* project. Non-walkable tags are not assigned any penalty, since they are not included in path calculations. By this means, agents tend to follow paths on better grounds, which allow them to move faster and through covered regions to avoid being detected by enemy surveillance. If they are covered by trees, the probability of the enemy detecting them via satellite images and aerial photos decreases. Once agents lose their moving ability by getting damaged below a threshold, they stop searching for paths to targets.

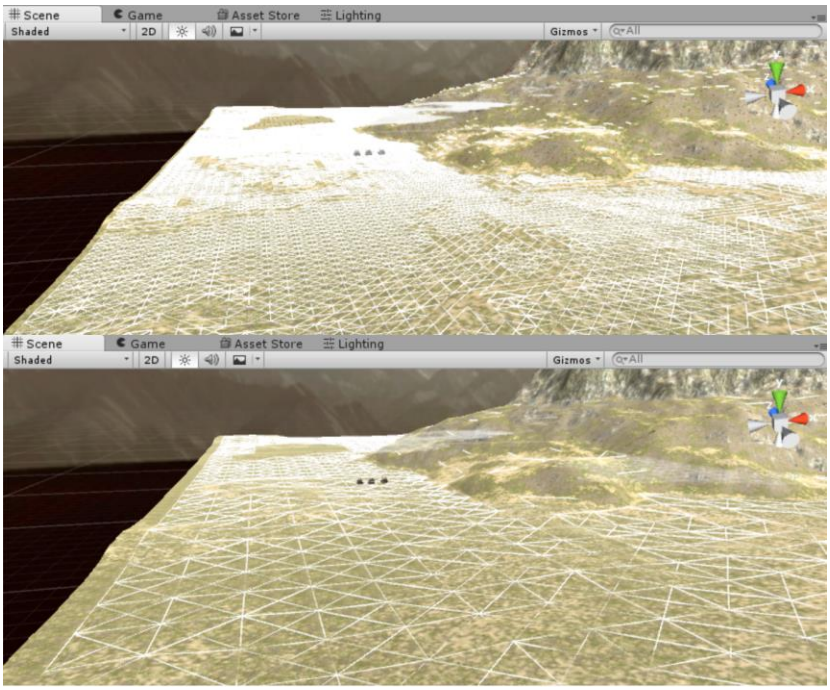
Figure 8. Penalties Assigned to Each Terrain Type



A* Pathfinding Project provides fast and asynchronous calculations. The main update loop for rendering a 3D environment does not stop and wait for function returns. The newest calculated paths are returned and assigned to agents as waypoint lists after they have been calculated via callback functions. Granberg's A* library also provides to make CPU intensive pathfinding calculations run in parallel on separate threads when supported by the hardware.

Another use of the A* algorithm is offline calculating paths between firing positions to store paths in arrays of three-dimensional vectors in a sequential order from start to the finish. These calculations are made on a grid graph (see Figure 9) that is less detailed than the usual agent pathfinding. Reducing the detail level of nodes causes a mismatch between the agent's path and particles path. The paths are not overlapping, which does not affect the results since a bunch of particles are traveling in groups meant to represent areas rather than precise locations. Additionally, storing fewer amounts of waypoints is advantageous, since they would be used on the GPU side as paths on which particles will wander. We optimized the number of waypoints on a laptop with an integrated graphic card Intel HD4400. A dedicated graphic card would be better and faster.

Figure 9. Main Grid Graph (above) versus. Less Detailed Grid Graph (below)



B. GPGPU

As discussed in Section **Error! Reference source not found.**, we used the aid of a GPU to calculate and update behaviors of 100,000 particles, which represents the possible positions of the blue forces on the battlefield. The game engine Unity 3D has a special component that provides access to “Compute Shaders” to make parallel processing computations easier. The game engine’s ease of use enables users to write their own scripts and attach them on game objects in the scene without bothering with update cycles during the game’s runtime. While creating the logic of our Compute Shader, we were inspired by the example of Michael Duncan’s blog (Duncan, 2014). The example is about a 10,000 stars galaxy simulation that runs 100,000,000 gravity calculations in every frame at 60 frames per second rate on the GPU.

The basic idea behind the Compute Shader is running update functions and other specific functions on the GPU side. When the simulation starts,

100,000 particles are generated as C# “structs” (see Figure 10). Every particle has eight fields totaling 64 bytes of data per particle. The array in which all the data particles are stored has a size of 6,400,000 bytes, which equals 6.4 megabytes. This is important information since transferring data between CPU and GPU causes delays depending upon the size of the data. In this case, it is not recommended that whole data be transferred back and forth.

Figure 10. Structure of a Particle

```
public struct Particle
{
public Vector3 position;
public Vector3 velocity;
public Vector3 color;
public Vector3 offset;
public float accelMagnitude;
public float weight;
public int pathNumber;
public int pathNodeCount;
};
```

Unity 3D style Compute Shader algorithms are similar to DirectX 11 DirectCompute technology, and are written in DirectX 11 style HLSL language. Next, we will inspect the Compute Shader we developed for the particle filter system that determines the highest probability placeholder on which the enemy is determined to be present. As seen in the first four lines in Figure 11, kernel names are defined with “#pragma” keywords. Afterwards, they will be defined as functions. “Particle.cginc” (line number 6) is the file that encloses particle data structure for the GPU side. “RWStructuredBuffer” type (line number 10-11) defines the buffer type that is both readable and writeable. On the other hand, “StructuredBuffer” type (line numbers 13 and 24) determines the read-only buffer type. Other variables (line numbers 8–32) are defined in the Compute Shader to be used later in the functions. Two functions (line numbers 34–49) return a random unsigned integer to be used again later in the process.

The UpdateParticles kernel, which is dispatched in every frame on every update of the whole cycle, can be seen in Figure 12. In line number 51, the number of threads is defined in three dimensions. For our Compute Shader, we have defined (128,1,1) number of threads, which we optimized by manipulating x, y, and z values of the three-dimensional thread vector on a laptop that runs an Intel HD4400 on-board graphic card. In this function, the velocity of the particles is calculated, and positions are updated. To avoid any collision between the particles and any other obstacles, every possible path combination is calculated and handed to the GPU side via the StructuredBuffer “path” (line number 13). Every particle has a variable of “pathNodeCount,” which is increased by one when the particle is close enough to the next waypoint in the list. Actually, rather than keeping an array of waypoint lists for every particle, particles read the next waypoint from the long array “path” via an indexing technique.

We will assume that the path with number 5 is assigned to a particle. The particle will look up the 250th number (five times the number of the longest path, which is determined by offline calculation and set 50 for this case), float3 (three-dimensional float, which represents the position in the scene), from the “path” array for the initial waypoint. When it gets close enough to that waypoint, it will increase pathNodeCount and will head to number 251, float3. Every particle has an offset distance to the center of the circle, within which 100,000 particles are uniformly spread. This circle defines an area of detection error. The positions of the particles are updated depending upon delta time, which is delivered to the Compute Shader from the CPU side on every update. When this kernel is dispatched from the CPU side, it is called for every particle in parallel threads in a much faster speed, compared to when it runs on the CPU.

Figure 11. Compute Shader Part-1 (Define Fields)

```
#pragma kernel UpdateParticles
#pragma kernel TotalWeight
#pragma kernel CullParticles
#pragma kernel ResetParticles

#include "Particle.cginc"

#define BLOCKSIZE   128

RWStructuredBuffer<Star> stars;
RWStructuredBuffer<Check> positionWeights;

StructuredBuffer<float3> path;

// time ellapsed since last frame
float deltaTime;

float cullRadius;

//position of culling
float3 cullPosition;

//for reset function
StructuredBuffer<int> possiblePaths;
float3 detectedPosition;

//max path length
static float max = 50;

//for random number generation
uint rng_state;

//faster
uint rand_lcg()
{
    // LCG values from Numerical Recipes
    rng_state = 1664525 * rng_state + 1013904223;
    return rng_state;
}

// better results but slower
uint rand_xorshift()
{
    // Xorshift algorithm from George Marsaglia's paper
    rng_state ^= (rng_state << 13);
    rng_state ^= (rng_state >> 17);
    rng_state ^= (rng_state << 5);
    return rng_state;
}
```

Figure 12. Compute Shader Part-2 (Update Particles)

```
051 [numthreads(BLOCKSIZE,1,1)]
052 void UpdateParticles(uint3 id : SV_DispatchThreadID)
053 {
054     uint i = id.x;
055     uint numStars, stride;
056     stars.GetDimensions(numStars, stride);
057 
058     float3 position = stars[i].position;
059     float3 velocity = stars[i].velocity;
060 
061     float3 A = path[stars[i].pathNumber*max+stars[i].pathNodeCount];
062     float3 B = path[stars[i].pathNumber*max+stars[i].pathNodeCount+1];
063 
064     if (distance (position,A+stars[i].offset)<3 && length(B)!=0 ){
065         stars[i].pathNodeCount++;
066     }
067 
068     velocity = A + stars[i].offset - position;
069     velocity = normalize(velocity) * stars[i].accelMagnitude;
070 
071     position += velocity * deltaTime;
072 
073     if (i < numStars)
074     {
075         stars[i].velocity = velocity;
076         stars[i].position = position;
077         stars[i].accelMagnitude = length(velocity);
078     }
079 
080 }
```

Every particle is updated in every frame. The UpdateParticles function is dispatched once from the update function of the game cycle (see Figure 13). Before launching threads on the GPU, the number of groups has to be determined. In this case, our block size is 128, and we determine our group number by dividing the particle number by the block size:

$$numberOfGroups = 100.000 / 128 \approx 781$$

When assigning a value to the block size, “occupancy” is the first consideration, which is active warps and maximum active warps ratio. We tried different values between 32 and 512 for the first dimension of block size, and 218 was the fastest one for our application. There are also occupancy calculators on

the web to find an optimum value for block size. Further information on this subject can be found on the web.

Figure 13. Dispatching and Launching Threads on the GPU

```
// bind resources to compute shader
particleCompute.SetBuffer(updateParticlesKernel, "stars", particleBuffer);
particleCompute.SetFloat("deltaTime", MyClock.DeltaTime * fastOrNormalPlay);
particleCompute.SetBuffer(updateParticlesKernel, "path", pathBuffer);
particleCompute.SetBuffer(updateParticlesKernel, "positionWeights", positionWeights);
// dispatch, launch threads on GPU
var numberOfGroups = Mathf.CeilToInt((float)startNumberAfterDetection / GroupSize);
particleCompute.Dispatch(updateParticlesKernel, numberOfGroups, 1, 1);
```

The second kernel in the Compute Shader is the “TotalWeight” function, which is dispatched when the enemy decides to shoot. This function updates the total probability on the checkpoints, which are pre calculated A* path waypoints, and the center positions of the firing points. It iterates through all the particles and increases the total probability of checkpoints that are closer than a specified distance by the particle’s probability. The challenge to increase a value on the shared memory begins when particles are trying to increment the value at the same time when they are being processed on parallel threads. To make it happen, thread operations should be synchronized. We used an “atomic” function InterLockedAdd function (see Figure 14, line number 102), which is provided as a DirectX 11 feature that handles adding a value to an integer or unsigned integer on the shared memory. The restriction of being able only to write on an integer or unsigned integer data forced us to use integer values for probabilities rather than a fractional number as floats. The third kernel decreases the probabilities of particles on a region where the enemy is supposed to take satellite images or aerial photos. Particles are initiated with a probability value of 5, which totals a probability of 500,000 of 100,000 particles. Particles at the center of the culled area are modified by decreasing their probability by 4. Depending upon the distance to the center point, the probabilities of other particles are lowered as well, until a minimum threshold of 0.1 is reached.

Figure 14. Compute Shader Part-3 (Total Weight)

```
[numthreads(BLOCKSIZE,1,1)]
void TotalWeight(uint3 id : SV_DispatchThreadID)
{
  uint i = id.x;
  uint numPos, stride;
  positionWeights.GetDimensions(numPos, stride);

  uint numStars, strides;
  stars.GetDimensions(numStars, strides);

  float radius = 20; //equals to grid node size
  float3 position = stars[i].position;

  if(i<numStars){

     [loop]
     for (uint j = 1; j < numPos; j++)
     {
        if (distance(position, positionWeights[j].position)<radius)
        {
           InterlockedAdd(positionWeights[j].weight, stars[i].weight);
        }
     }
  }
}
```

Figure 15. Compute Shader Part-4 (Cull Particles)

```
[numthreads(BLOCKSIZE, 1, 1)]
void CullParticles(uint3 id : SV_DispatchThreadID)
{
    uint i = id.x;
    uint numStars, stride;
    stars.GetDimensions(numStars, stride);

    float3 position = stars[i].position;
    float weight = stars[i].weight;
    float dist = distance(cullPosition, position);
    if (dist <cullRadius) {

        float decrease = clamp(round(cullRadius /dist),1,4);
        weight -= decrease;
        if (weight<0) weight = 0.1;
    }
    if (i < numStars) {
        stars[i].weight = weight;
    }
}
```

The last kernel (see Figure 16) in the Compute Shader is responsible for resetting the particles as soon as the howitzer platoons are detected. Usually, particles are transferred to the detected position within an error circle. On the

CPU side, enemy AI reasons and attaches possibilities to the nearby firing position where enemy thinks blue forces will move next. To transfer this data to the GPU side, we created an array of length 100 that consists of firing positions IDs, and which has repeated numbers of these IDs depending upon the possibility assigned to them. On the GPU side, particles are assigned paths that lead to specified firing positions by simply choosing one member randomly from the array of length 100. Since randomizing on Shaders is not as easy as it is on a typical script, we used Nathan Reed's method (see Figure 11, lines 30–49) that was discussed in his blog (Reed, 2012).

Figure 16. Compute Shader Part-4 (Reset Particles)

```
129 [numthreads(BLOCKSIZE,1,1)]
130 void ResetParticles(uint3 id : SV_DispatchThreadID)
131 {
132     uint i = id.x;
133     uint numStars, stride;
134     stars.GetDimensions(numStars, stride);
135 
136     //size of possiblepaths
137     uint size = 100;
138 
139     rng_state = id.x;
140 
141     int pathNumber;
142     float3 position = stars[i].position;
143 
144     position = detectedPosition + stars[i].offset;
145 
146     // Generate a random float in [0, 1)...
147     float f0 = float(rand_xorshift()) * (1.0 / 4294967296.0);
148 
149     f0 = f0 * size;
150     pathNumber = possiblePaths[round(f0)];
151 
152     if (i < numStars)
153     {
154         stars[i].position = position;
155         stars[i].pathNumber = pathNumber;
156         stars[i].weight = 5;
157         stars[i].pathNodeCount = 1;
158     }
159 }
```

C. SPEEDING UP THE SIMULATION

We present two modes for the main simulation: real-time mode and experiment mode. Real-time mode is the 3D visualization of our survivability

model, whereas experiment mode runs without rendering any image, and is focused on multiple runs that are optimized to be executed faster. The main method has been discussed in Chapter III. In this chapter, provide details about the algorithm.

Unity has the ability to run the application in batch mode, simply by calling via a command line. It is not using the GPU for rendering issues, but executes the main game cycles under the hood. When the V-sync option is disabled, we determined that our application was able to run 4,000–10,000 frames per second. This is a significant increase, but it does not make sense when a delta time variable is used as a factor for movement. Because delta time values will drop to smaller values, agents would cover the same distance as in the real-time version due to a dependence upon the time that passes between two update cycles. Making our simulation frame dependent, we added another mode to the simulation that replaced the delta time with a fixed-time step. Therefore, having more frames increased the speed of the simulation. To increase every object's behavior, we added the time factor into every movement algorithm. Another struggle was speeding up the physics calculation. Physics calculations are calculated outside of the main update cycle. They are updated at every fixed-time step, which can be manually set via the Unity Time Manager. Its default value is 20 milliseconds, meaning that updates are executed 50 times a second regardless of the FPS rate. Due to this, we excluded physics from the simulation. There are no colliders and agents can walk through obstacles if their path passes through them. However, this also seems impossible for our grid node based A* calculations. Additionally, agents are aligned with the terrain by casting a ray from a distance above them, finding the normal of the ray-intersected surface, and adjusting translation and rotation of the agents.

Other than movement, there are lots of functions dependent upon the time. Many events and functions are executed after some delay, such as waiting for reload time of the weapons. Unity uses coroutines to execute a block of code during updates over a specified time. It is easy to implement a time delay via

coroutines. As seen in Figure 17 line number 8, an instance of WaitForSeconds class is instantiated to cause a return null for seconds determined as the argument. However, these seconds are real-time seconds, so we need an algorithm to decrease these seconds proportionately to the time scale.

Figure 17. Simple Co-Routine Example

```
001  public void doSupply()
002  {
003     StartCoroutine("supplyingProcess");
004  }
005
006  IEnumerator supplyingProcess()
007  {
008      yield return new WaitForSeconds(10);
009      myFSM.SendEvent("FinishedSupply");
010  }
```

To regulate the time properly, we created our own clock logic (see Figure 18). We are using MyClock.time value for delta time, which is be the time between two frames throughout the whole simulation. Depending upon the type of the simulation (experiment mode or real-time mode), this value is set to the proper value on every update cycle (see Figure 28, lines number 46–55). If the simulation has been started in the real-time mode, our MyClock.time is simply set equal to Time.deltatime value, which is calculated in each frame by Unity Engine. If the other simulation has been started in the experiment mode, the MyClock.time value is fixed on a time step that is set at the beginning of the simulation to a specified value.

As discussed in Section F, the main struggle when speeding up a waypoint dependent agent movement is the oscillating around a waypoint during the agent movement. This is a big step, and it can never get closer than the threshold distance to the target waypoint when it sets its target to the next waypoint. We first made experiments and sought for a secure and maximum time step, which speeds the movement without having weird behaviors by overshooting waypoints. With this secure time step, there is a secure distance calculated for agents traveling between frames. For instance, fixing MyClock.time

to 0.1 seconds, means that in each frame, 100 ms have been assumed to pass, and depending upon this time interval, we can calculate the distance agents should have covered during that time. After numerous tests, we are able to fix the time steps in the experiment mode to 100ms (0.1 seconds). Taking it into account that in our real-time simulation we achieved a frame rate of 60–90 FPS —equal to having a delta time value between 16ms and 11ms—we are speeding up the simulation by a rate of approximately 6–16 times in just one frame.

Changing the Unity's internal stopwatch to increase the time in real-time seconds, we created our WaitForSeconds class that keeps waiting relative to our new clock (see Figure 19, line numbers 5–11). It is also able to track time as Unity's time values when the simulation is started in real time. Using these methods, we were able to switch simulation time logic between real time and experiment time by simply setting a Boolean value (see Figure 18, line number 48). These modifications were enough to speed up the simulation. Particles whose movement and probability updates are made on the GPU side, were speeding up with proper rate, since in each frame we were handing the MyClock.Time value as the delta time to the GPU side.

As an additional feature, we put a slider setting on the menu, where the user can increase the speed of the simulation up to 5 times using the slider bar (see Figure 20). On the other hand, when the simulation is started in the experiment mode, there is no visual output rendered to be seen by the user. Instead, the simulation runs under the hood and writes to a CSV file by outputting the survival times after every design point run.

Figure 18. MyClock Lass

```
001     public class MyClock : MonoBehaviour
002     {
003         #region private fields
004         private static bool experimentMode;
005         private static float deltaTime;
006         private static float time;
007         #endregion
008
009         #region public fields
010         public static bool experimentModeOn;
011         /// <summary>
012         /// set experimental deltaTime in unity editor
013         /// </summary>
014         public float myDeltaTime;
015         #endregion
016
017         #region properties
018         public static bool ExperimentMode
019         {
020             get { return experimentMode; }
021         }
022         public static float DeltaTime
023         {
024             get { return deltaTime; }
025             //to be able to pause it pausetoggle script
026             set { deltaTime = value; }
027         }
028         public static float MyTime
029         {
030             get { return time; }
031         }
032         #endregion
033
034         void Awake()
035         {
036             time = 0f;
037
038             experimentModeOn = ExperimentRoot.experimentMode;
039             experimentMode = experimentModeOn;
040             deltaTime = myDeltaTime;
041
042             if (!experimentMode)
043                 deltaTime = Time.deltaTime;
044         }
045
046         void Update()
047         {
048             if (experimentMode)
049                 time += deltaTime;
050             else
051             {
052                 deltaTime = Time.deltaTime;
053                 time = Time.time;
054             }
055         }
056
057     }
```

Figure 19. Modified WaitForSeconds

```
001     public class MyWaitForSeconds: CustomYieldInstruction
002     {
003         private float waitTime;
004 
005         public override bool keepWaiting
006         {
007             get
008             {
009                 return MyClock.MyTime < waitTime;
010             }
011         }
012 
013         public MyWaitForSeconds(float delayTime)
014         {
015             waitTime = MyClock.MyTime + delayTime;
016         }
017     }
```

Figure 20. Time Slider in the Enu

THIS PAGE INTENTIONALLY LEFT BLANK

V. ANALYSIS OF THE EXPERIMENT

Adding the ability to run the simulation without rendering via a command line in experiment mode, enables us to fast-forward actions and movement. Therefore, multiple runs in a shorter amount of time could be successfully executed, allowing us to conduct more experiments. Moreover, we also provide users with the opportunity of inputting their own designs of experiments via a CSV file. This turns the simulation into an experiment tool to test different influences and effects on the desired response variable. To highlight this ability of the simulation, we conducted an example of an experiment that inspects the effects of five factors on the survival time.

A. DESIGN OF THE EXPERIMENT

The experiment we have designed for the purpose of our research focuses on the factors that affect the survivability of the modern artillery units while they are providing fire support in an assault mission. We created the ASM to visualize the main survivability movements on the terrain, and created an experiment tool that is able to make multiple runs with provided design points. To achieve multiple runs with the simulation in a shorter time we developed a method to speed up the simulation time (see Section F).

The response variable for the experiment is the survival time of the first platoon (three howitzers). The time when all of three howitzers of the platoon lose their moving ability or are destroyed is recorded as the response variable for the simulation. Since destroying moving howitzer units by indirect fire is a hard task to accomplish, we assumed that all members of the platoon losing their moving capability would cause the platoon to stop operations and make them vulnerable to artillery fire on open terrain.

We tested five factors for this experiment. It is also possible to test other factors by easily changing variables in scripts. Here we provided five main factors; however, anyone familiar with C# language could review the code and

manipulate the factor variables. Our intent was to create a user-friendly interface that would make it easier to change experiment factors; however, the time constraints of the research did not for development time. For this study, our factors include:

- A - Movement Speed of Howitzer
- B - Training Level of Howitzer Units
- C - Enemy Radar Quality
- D - Effect of the Enemy Rounds
- E - Armor Thickness of Howitzer Units

The experiment will be executed in the two-level (low and high) fractional factorial design. In this case, it will be one-half fraction and resolution V design (2_V^{5-1}). The generator for the fifth factor (E) will be ABCD. There will be 16 design points, and 30 simulation replicates for each design point. Design points with a determined factor list can been in Table 2.

We provided a CSV file that includes these factors in Table 2 in the relative location “\asm_Data\StreamingAssets\dPFile.csv” of the executable simulation file. Once the simulation starts, it parses this document and runs specified multiple times at every design point with designated factor values. This file can be edited before the simulation execution and desired factor values can be set for each design point. The “StreamingAssets” folder also has the “config.csv” file, where the user can set the number of multiple runs.

The Unity standalone build stores an execution log in the text file “\[executable_file_name]\output_log.txt.” This file also includes debug logs and console outputs. We provided an extra organized output text file that stores data regarding each un in the identified folder location “\[executable_file_name] \StreamingAssets\experimentOutputs.csv.” The simulation adds every output into this file with the execution time info. It is not overwritten on each execution; therefore, the user can track old outputs when examining this file.

For this experiment, two levels (low and high) are set for each factor (see Table 3). Movement speed (5–10) is the distance in scene graph units that howitzers take in one second. Training level (10-3) determines the quality of the training of the howitzer crews. In the simulation, it is modeled as the preparation time to shoot one round in a specified number of seconds. A higher value means more time to spend to reload and to prepare before shooting, which causes units to stay more static on a firing position and to be more vulnerable to indirect fires. Enemy radar quality (3-1) affects the total time between blue units shoot from a position and enemy detects that position on its radar. Radar quality value is a multiplier of this detection time. The higher the radar quality, the larger is the detection time.

Table 2. Design Points

Run	A	B	C	D	E=ABCD	Treatment Combination
1.	-	-	-	-	+	e
2.	+	-	-	-	-	a
3.	-	+	-	-	-	b
4.	+	+	-	-	+	abe
5.	-	-	+	-	-	c
6.	+	-	+	-	+	ace
7.	-	+	+	-	+	bce
8.	+	+	+	-	-	abc
9.	-	-	-	+	-	d
10.	+	-	-	+	+	ade
11.	-	+	-	+	+	bde
12.	+	+	-	+	-	abd
13.	-	-	+	+	+	cde
14.	+	-	+	+	-	acd
15.	-	+	+	+	-	bcd
16.	+	+	+	+	+	abcde

Table 3. Experiment Factor Values

Factor Name	Low Level	High Level
A –Movement Speed	5	10
B – Training Level	10	3
C – Enemy Radar Quality	3	1
D – Effect of Enemy Rounds	30	60
E – Armor Thickness	100	150

Effect of the enemy round (30–60) is the damage value of one round on the impact point. The damage decreases as the distance between the howitzers and the impact point grows (area damage). Armor thickness (100–150) is the starting health point of the howitzers, which is also the determinant of no-move and destroyed states. This determinant decreases with time depending upon the shots taken during the battle.

B. ANALYSIS OF THE OUTPUTS

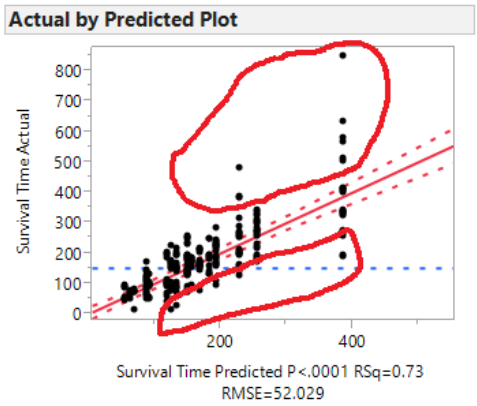
After running the simulation 480 times (30 multiple-runs x 16 design points), and recording response variables and the time at which all of the three howitzers lose their moving ability, we used the software JMP Pro 11 to analyze the output and conduct a regression analysis. The JMP screening procedure was used to list matching design points, and a fit model was created with main factors and two factor interactions.

When we ran our first model with the main factors and two factors interactions, we received a fitted model that was able to explain 73% of the variability (see Table 4). There were 19 extreme outliers that are displayed in Figure 21, circled in red.

Table 4. Summary of Fit of First Run

RSquare	0.729779
RSquare Adj	0.721044
Root Mean Square Error	52.02946
Mean of Response	150.3309
Observations (or Sum Wgts)	480

Figure 21. Actual by Predicted Plot of First Run



Next, we used JMP stepwise regression methods to find a best fit for our model. From different provided methods in the software, we chose the “minimum Bayesian Information Criterion” method to design the model. This method excludes the factors and interactions that are insignificant and create the best-fit model by simply clicking a button. The best-fit model excluded the following four two factor interactions from the model:

- Movement Speed * Effect of Enemy Rounds
- Training Level * Effect of Enemy Rounds
- Enemy Radar Quality * Effect of Enemy Rounds
- Training Level * Enemy Radar Quality

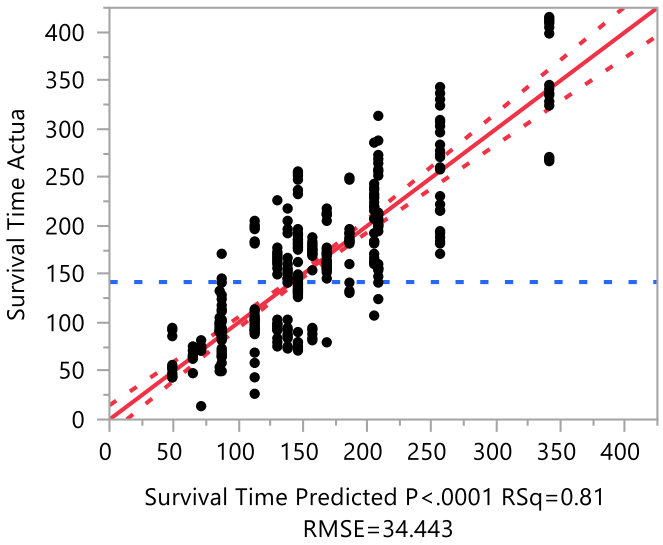
Additionally we excluded the 19 extreme values that were far outliers, and made the model with 461 run outputs. Running the fit model, we have “Actual by

Predicted Plot" as an output where we can see how well our predicted model fits the actual data (see Figure 22). Table 4 shows us the summary of fit. We have an RSquare of 81%, which indicates that the model explains 81% of variability of the survival time around its mean. Our adjusted RSquare differs slightly from the RSquare, which shows that almost every independent variable in our model is affecting our response variable.

Table 5. Summary of Fit

RSquare	0.813154
RSquare Adj	0.808576
Root Mean Square Error	34.44337
Mean of Response	141.8946
Observations (or Sum Wgts)	461

Figure 22. Actual by Predicted Plot



JMP's Analysis of Variance report (see Table 6) compares our fitted model with a model where all the predicted values are the same as the mean of the response variable. We see that ours is significantly different than this ideal model, since we already determined that ours is capable of explaining 81 % of the variability. Having a small p-value below the threshold, means also that there is at least one significant factor in our model.

Table 6. Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	11	2318172.3	210743	177.6404
Error	449	532669.3	1186	Prob > F
C. Total	460	2850841.6		<.0001*

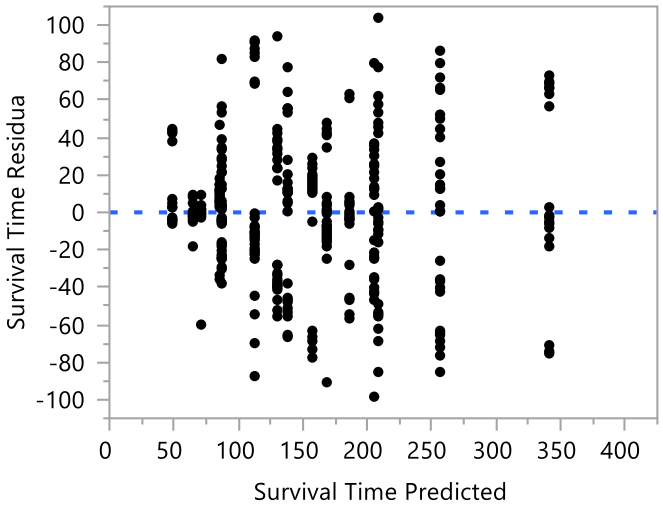
The Lack of Fit report explains how well our model fits the data. From the p-value (0.2728) in Table 7, we see that there is no significant lack of fit in our model. The model's independent variables (our five factors) managed to fit the data well without requiring any additional independent variables.

Table 7. Lack of Fit

Source	DF	Sum of Squares	Mean Square	F Ratio
Lack Of Fit	4	6108.31	1527.08	1.2905
Pure Error	445	526560.94	1183.28	Prob > F
Total Error	449	532669.25		0.2728
				Max RSq

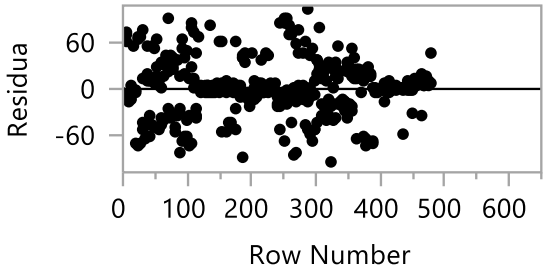
The Residual by Predicted Plot (see Figure 23) seems to meet the constant variance assumption by not having a vertical width of the scatter, which increases or decreases across the plot. The errors are randomly scattered around zero, and the mean is close to zero.

Figure 23. Residual by Predicted Plot



The Residual by Row Plot (See Figure 24) shows the residuals plotted by the row number, which indicates our observation number as well. There is no specific pattern, and dots seem to be scattered randomly, which ensures that our independence assumption has been met.

Figure 24. Residual by Row Plot



The distribution of the residuals is normal (see Figure 25). The histogram is unimodal and symmetric. On the normal quantile plot, we see an S shape where residuals contain larger values around zero (left-of-zero and right-of-zero). This satisfies the normality assumption. We also see a number of outliers that are darkened as seen on the box plot. Outliers are still within the 3 IQR (Interquartile Range, which equals the central rectangle range, starts from first quartile, and ends at third quartile), above the third quartile, and the 3 IQR below

the first quartile. They can be defined as suspected outliers (outside values), rather than outliers (far outside values).

Figure 25. Residual Survival Time

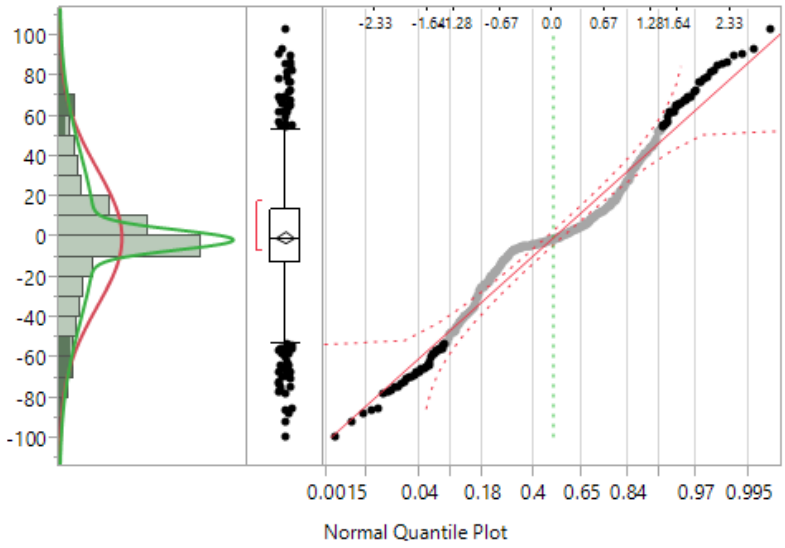


Table 8. Sorted Parameter Estimates

	Term	Estimate	t Ratio	Prob>|t|
1	Armor Thickness	54.198327		<.0001*
2	Training Level	-38.19359		<.0001*
3	Movement Speed	-22.71931		<.0001*
4	Enemy Radar Quality	-15.26849		<.0001*
5	Movement Speed*Armor Thickness	-14.26765		<.0001*
6	Training Level*Armor Thickness	-13.48685		<.0001*
7	Effect of Enemy Rounds	-12.28807		<.0001*
8	Movement Speed*Training Level	8.9139336		<.0001*
9	Enemy Radar Quality*Armor Thickness	-7.089428		<.0001*
10	Effect of Enemy Rounds*Armor Thickness	-4.625371		0.0043*
11	Movement Speed*Enemy Radar Quality	3.511009		0.0300*

Our reduced model shows that every factor used in the fit model is significantly important (see Table 8). P-values of every term are below the significance level 0.05. In Table 8, these terms are listed in the order of decreasing significance. At the end of the experiment, we concluded that the most significant effect on survival time was the armor thickness. The training level of the crew was in the second place, which is respectively followed by the movement speed and enemy radar quality with closer significance rates. One interesting result was that the effect of the enemy rounds was in the seventh order after two two-factor interactions, but was still within a high significance ratio.

The resulting model is:

$$\text{Predicted Survival Time} = 147.56 + 54.20\ X_{\text{Armor Thickness}} - 38.19\ X_{\text{Training Level}} - 22.72\ X_{\text{Movement Speed}} - 15.27\ X_{\text{Enemy Radar Quality}} - 14.27\ X_{\text{Movement Speed}}\ X_{\text{Armor Thickness}} - 13.49\ X_{\text{Training Level}}\ X_{\text{Armor Thickness}} - 12.28\ X_{\text{Effect of Enemy Rounds}} + 8.91\ X_{\text{Movement Speed}}\ X_{\text{Training Level}} - 7.10\ X_{\text{Enemy Radar Quality}}\ X_{\text{Armor Thickness}} - 4.63\ X_{\text{Effect of Enemy Rounds}}\ X_{\text{Armor Thickness}} + 3.51\ X_{\text{Movement Speed}}\ X_{\text{Enemy Radar Quality}}$$

VI. CONCLUSION

The simulation “Artillery Survivability Model (ASM)” is the product of this study that seeks answers to four research questions (RQ). ASM succeeded to generate an adequate algorithm for artillery survivability movements (RQ1) by creating a 3D virtual environment presenting a restricting battlefield. As we did get into detail in previous chapters, besides providing a reasonable position selection from all the determined firing positions that are generated by terrain processing, agents in the simulation use covered paths calculated by A* algorithm, when they move to the next firing position.

As a solution for creating an indirect fire threat map (RQ2) for artillery, we used particle filters. These filters consisted of 100,000 particles, all of which contained the properties of probability for the positions of the blue forces, including speed, velocity, color, and target. When each of the units was detected, particles appeared on that location and began to spread to the next potential firing positions. They also changed probability and color on locations where the enemy was able to take visual reconnaissance. In real-time mode, we visualized this occupancy and threat map with particles that were actually begin updated on GPU, using a Compute Shader component of Unity.

The simulation enabling the user to design experiments and to execute multiple runs for each design mode, automatically made it possible to test factors on survival time (RQ3). In Chapter V, we provided an example experiment inspecting the five main factors that affect the survival time of the artillery. Our goal was to test the random selection process that agents do at some proportion of the time when they do not select firing positions with scoring methods, because this randomness would help them to follow random patterns and provide unpredictability. To test this, we developed a learning algorithm for the enemy that adapts to feedbacks on failure or success at guessing the position of the blue forces, and improves its guess function by online learning. Because of the

time constraint and the broad scope of the study, we were not able to implement this method, but will add this task to our future work.

As the experiment mode provided insights about factors through experimentation, the real-time rendered simulation visualized the survivability movements on the real terrain. The user was able to compare the different behaviors of two howitzer platoons that were fighting side-by-side. This visualization could provide insights into survivability movement and an easy understanding of the concept for inexperienced artillery officers. The simulation could also be used as a training tool (RQ4) to compare the results of an artillery officer who worked on a 2D map and determined appropriate firing positions. Nevertheless, the simulation needs an upgrade to provide the user with input of chosen firing positions, and to provide a digital comparison between the input and agent's behavior of position selection. This feature will also be included in future work.

As a result, we were able to create a 3D agent-based simulation to visualize platoon-level artillery operations with high-end graphics, and also to provide multiple-run experiments in experiment mode up to 50 times faster than the real-time simulation. The experiment mode enabled users to create their own design and to experiment on the factors in which they were interested. In Chapter IV, we provided an example for an experiment whose outputs were reasonably interpreted.

Using Compute Shaders to run on the GPU side, it was possible to develop particle filter position tracking algorithms, which provided a more reasonable and realistic AI for the enemy. During the experiment mode, when we called the application via the command line with the argument "–batchmode," Unity was not using the graphic card for both rendering and Compute Shaders. Therefore, we were not able to use particle filter algorithms for enemy tracking during this mode. We swapped the particle filter algorithm with a simpler algorithm that shot to the current position of the blue forces with some random error.

Developing a unique method for survivability movement will help the artillery units to change position more systematically, by considering terrain features, and using the closest covered path to the set destination. In addition, to fill the gap of missing documentation regarding how to maneuver within the position area, this simulation tool will facilitate decision making in an even shorter time. Considering that advising the next firing position to the platoon leader is the job of one person in the POC, and this person tends to follow similar patterns, one of the other advantages of the simulation is that it has an algorithm, which provides random outputs for firing positions.

A. ABOUT THE PLATFORM

Using a popular game engine to create a simulation, as well as Unity's convenient user interface, and the power of the C# language assisted in the development process. Assets, code libraries, and models that users can buy or download for free from the Unity's Asset Store, formed the base architecture of the simulation. The quality of their architecture also made it possible to import these libraries and use them efficiently to write code for our purposes. Learning and mastering the game engine is easy if the developer has a background in the basic game development.

The user network of Unity is very wide and interactive. The interactive use of forums and sharing of user information over the web makes research quite easy. Also, social networks facilitate communicating with people who are main developers of the Unity game engine. For instance, we were having problems using Compute Shaders in batch mode (experiment mode). Consulting the Unity developer of Compute Shaders via Twitter, provided us with quick assistance.

One difficulty that we confronted was in finding solutions to our experimental mode. Unity is a game engine and focuses on real-time rendering solutions. They also support people who are developing serious games and simulations for military solutions. Although our attempt to convert the application into an experimental mode without rendering any image was successful, it is

almost impossible to find any other similar solution on the web. The first intent to create batch mode was for server side applications and network solutions, which do not require rendering.

B. FUTURE WORK

Adding randomness to the decision of the platoon leader when choosing next firing position was an idea about preventing and making it difficult for the enemy to learn blue forces' position changing algorithm. We were interested in the effect of this randomness on the survival time; however, without making the enemy learn, adapt, and improve its guess functions, it is impossible to inspect the randomness effect. As a future work, enemy AI may be improved to an adaptable function so that it can use online learning algorithms to make better guesses about the next position of the blue forces. Furthermore, an experiment may be run with the tools we provided to examine the effect of the randomness on the survival time.

A user interface could be developed for experiment mode. With this edition of the simulation, users should review code to change the main factors of the experiment. The design of experiment CSV file enables users to create their own designs for experimenting with five pre-determined factors (Armor thickness, speed, enemy radar quality, effect of the enemy round, and training level). Another user interface could be developed for the real-time simulation to let users input their pattern for survivability movement and then compare their own work with the simulation's output.

The simulation can be upgraded to a battalion level, which would make it possible for battalion level operations to be inspected by running experiments. Interactions between batteries can be modeled. Additionally, communication with the headquarters and maneuver units can be modeled, which is a significant factor for fire support missions. Another option would be to create an interactive version of the simulation where the user would be able to assign future firing positions for the howitzer platoon and compare results with the agent's decisions.

LIST OF REFERENCES

Darken, C. J., & Anderegg, B. G. (2008). Particle filters and simulacra for more realistic opponent tracking. *AI Game Programming Wisdom 4*, 419–428.

Department of the Army. (2000). *Tactics, techniques, and procedures for M109A6 Howitzer (Paladin) Operations* (FM 3-09.70). Washington, DC: Author.

Department of the Army. (2001). *Tactics, techniques and procedures for the Field Artillery Battalion* (FM 3-09.21). Washington, DC: Author.

Department of the Army. (2014). *Field artillery operations and fire support* (FM 3-09). Washington, DC: Author.

Duncan, M. A. (2014, 02 01). Micky D's random thoughts. Retrieved 05 2016, from N-Body Galaxy Simulation using Compute Shaders on GPGPU via Unity 3D: https://mickyd.wordpress.com/2014/02/01/n-body-galaxy-simulation-using-compute-shaders-on-gpgpu-via-unity-3d/.

Fann, C. M. (2006). Development of an artillery accuracy model (Master's thesis). Retrieved from Calhoun http://hdl.handle.net/10945/2500.

Fish, C. B., & Murray, M. V. (2008, May-June). The improved firefinder position analysis system. *Fires*, 30–33.

Guzik, D. M. (1988). A Markov model for measuring artillery fire support effectiveness (Master's thesis). Retrieved from Calhoun http://hdl.handle.net/10945/23070.

Heisinger, R. R. (2007). Optimization of Marine Corps Artillery Battalion supply distribution network (Master's thesis). Retrieved from Calhoun http://hdl.handle.net/10945/3297.

Kang, C. (1995). A cost and operational effectiveness analysis for future artillery system in Korea (Master's thesis). Retrieved from Calhoun http://hdl.handle.net/10945/7461.

Klein, G. A., Calderwood, R., & Macgregor, D. (1989). Critical decision method for eliciting knowledge. *IEEE Transactions On Systems, Man, And Cybernetics*, *19*(3), 462-472.

Reed, N. (2012, January). Nathan Reed's Blog. Retrieved April 2016, from Quick and Easy GPU Random Numbers in D3D11: http://www.reedbeta.com/blog/2013/01/12/quick-and-easy-gpu-random-numbers-in-d3d11/.

Schneider, M. W., & Ferrara, A. R. (1989). A Command and Control wargame to train officers in the integration of tactics and logistics in a field artillery battalion (Master's thesis). Retrieved from Calhoun http://hdl.handle.net/10945/26925.

Schuller, D. (2011). *C# game programming: For serious game creation*. Boston: Course Technology.

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California